Cost-Filtering Algorithms for the two Sides of the *Sum of Weights of Distinct Values* Constraint

Nicolas Beldiceanu[*], Mats Carlsson[*], and Sven Thiel[+]

[*]SICS, Lägerhyddsvägen 18, SE-75237 Uppsala, Sweden
{nicolas,matsc}@sics.se
[+]MPI für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
sthiel@mpi-sb.mpg.de

October 11 2002
SICS Technical Report T2002:14
ISRN: SICS-T–2002:14-SE
ISSN: 1100-3154

Abstract. This article introduces the *sum of weights of distinct values* constraint, which can be seen as a generalization of the *number of distinct values* as well as of the *alldifferent*, and the *relaxed alldifferent* constraints. This constraint holds if a cost variable is equal to the sum of the weights associated to the distinct values taken by a given set of variables. For the first aspect, which is related to domination, we present four filtering algorithms. Two of them lead to perfect pruning when each domain variable consists of one set of consecutive values, while the two others take advantage of holes in the domains. For the second aspect, which is connected to maximum matching in a bipartite graph, we provide a complete filtering algorithm for the general case. Finally we introduce several generic deduction rules, which link both aspects of the constraint. These rules can be applied to other optimization constraints such as the *minimum weight alldifferent* constraint or the *global cardinality* constraint with costs. They also allow taking into account external constraints for getting enhanced bounds for the cost variable. In practice, the *sum of weights of distinct values* constraint occurs in assignment problems where using a resource once or several times costs the same. It also captures domination problems where one has to select a set of vertices in order to control every vertex of a graph.

Keywords: Constraint Programming, Global Constraint, Cost-Filtering, Assignment, Domination, Bipartite Matching.

Cost-Filtering Algorithms for the two Sides of the *Sum of Weights of Distinct Values* Constraint

Nicolas Beldiceanu[*], Mats Carlsson[*], and Sven Thiel[+]

[*]SICS, Lägerhyddsvägen 18, SE-75237 Uppsala, Sweden
{nicolas,matsc}@sics.se
[+]MPI für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
sthiel@mpi-sb.mpg.de

Abstract. This article introduces the *sum of weights of distinct values* constraint, which can be seen as a generalization of the *number of distinct values* as well as of the *alldifferent*, and the *relaxed alldifferent* constraints. This constraint holds if a cost variable is equal to the sum of the weights associated to the distinct values taken by a given set of variables. For the first aspect, which is related to domination, we present four filtering algorithms. Two of them lead to perfect pruning when each domain variable consists of one set of consecutive values, while the two others take advantage of holes in the domains. For the second aspect, which is connected to maximum matching in a bipartite graph, we provide a complete filtering algorithm for the general case. Finally we introduce several generic deduction rules, which link both aspects of the constraint. These rules can be applied to other optimization constraints such as the *minimum weight alldifferent* constraint or the *global cardinality* constraint with costs. They also allow taking into account external constraints for getting enhanced bounds for the cost variable. In practice, the *sum of weights of distinct values* constraint occurs in assignment problems where using a resource once or several times costs the same. It also captures domination problems where one has to select a set of vertices in order to control every vertex of a graph.

1 Introduction

It has been quoted in [7] that an essential weakness of constraint programming is related to optimization problems. This first means that very often the lower bound of the cost to minimize is quite poor and in addition there is usually no back-propagation from the maximum allowed cost to the decision variables of the problem. This was especially true when the total cost results from the addition of different elementary costs. For these reasons, several authors have started to reuse methods from operations research for tackling this problem. This was for instance done within scheduling in [1] as well as for assignment problems in [3] and for the maximum clique problem in [6].

The purpose of this article is to contribute to this line of research by considering a new kind of cost-function which arises in quite a lot of practical assignment and covering problems but for which neither a direct model[1] nor a filtering algorithm was available. A second contribution of this article is to come up with new generic deduction rules, which can also be applied for improving the deductions performed by existing constraints using cost filtering techniques. In particular, this holds for a generalization of the *global cardinality constraint with costs* [13] and for a generalization of the *assignment* constraint with costs [7], [14]. In addition these rules allow taking into account external constraints for getting better bounds for the cost variable (e.g. better bound for the cost of the *minimum weight all different* constraint with a restriction on the maximum number of cycles [4]).

The constraint introduced in this article has the form $\text{sum\_of\_weights\_of\_distinct\_values}(Assignments, Values, Cost)$, where:

- $Assignments$ is a collection of n items where each item has a var attribute; var is a domain variable[2] which may be negative, positive or zero.
- $Values$ is a collection of m items where each item has a val as well as a $weight$ attribute; val is an integer which may be negative, positive or zero, while $weight$ is a non-negative integer. In addition, all the val attributes should be pairwise distinct. $\mathcal{Values}$ denotes the set of values taken by the val attributes.
- $Cost$ is a domain variable which takes a non-negative value.

The items of a given collection are bracketed together; for each item we give its attributes as a pair $name-value$ where $name$ and $value$ respectively designate the name of the attribute and its associated value.

The $sum\_of\_weights\_of\_distinct\_values$ constraint holds if all the variables of $Assignments$ take a value in $\mathcal{Values}$ and if $Cost$ is the sum of the $weight$ attributes associated to the distinct values taken by the variables of $Assignments$. For instance, the following constraint $\text{sum\_of\_weights\_of\_distinct\_values}\left(\begin{array}{l}\{var-1, var-6, var-1\},\\ \left\{\begin{array}{ll}val-1 & weight-5,\\ val-2 & weight-3,\\ val-6 & weight-7\end{array}\right\}, 12\end{array}\right)$ holds since the cost 12 is the sum of the weights 5 and 7 respectively associated to the two distinct values 1 and 6 occurring in $\{var-1, var-6, var-1\}$. Observe that the $sum\_of\_weights\_of\_distinct\_values$ constraint is different from the *minimum weight all different* constraint [14] and from the *global cardinality constraint with costs* [13] since these two constraints compute the overall cost from a cost matrix, which for each variable-value pair gives its corresponding contribution in the cost.

Since we don't presume any specific use of the $sum\_of\_weights\_of\_distinct\_values$ constraint, this article assumes that the domain of the $Cost$ can be restricted in any way. Concretely, this means that we want to be able to prune the assignment variables according to the minimum and maximum values of the $Cost$ variable, as well as according to any holes. In contrast to most previous work [3], [7] where algorithms

[1] A model for which, beside the cost and the assignments variables, no extra variables have to be introduced.

[2] A domain variable is a variable that ranges over a finite set of integers.

from operation research could be adapted, we had to come up with new algorithms for performing these tasks.

Sect. 2 generalizes the filtering algorithm presented in [2] for handling the number of distinct values constraint to a complete[3] algorithm for the case where each domain of an assignment variable consists of one single interval of consecutive values and where all the weight are equal to one. It also provides several deduction rules, which take partially into account holes in the domains. Sect. 3 introduces a lower bound for the sum of the weights of the distinct values as well as an algorithm for evaluating this lower bound. Sect. 4 defines the notion of *lower regret* associated to a given value and provides the corresponding filtering algorithm, which propagates from the maximum allowed cost to the assignment variables. Sect. 5 presents a tight upper bound of the sum of the weights of the distinct values, while Sect. 6 introduces the notion of *upper regret* as well as an algorithm which propagates from the minimum allowed cost to the assignment variables. Sect. 7 presents several generic deduction rules which combine the lower or the upper regret as well as the domain of the cost variable or some additional constraints on the assignment variables. Finally, Sect. 8 situates the *sum_of_weights_of_distinct_values* constraint among existing constraints and shows how domination problems as well as some assignment problems, like the warehouse location problem [15] fit into this constraint.

Before starting, let us first introduce the notations used throughout this article.

Notations and Conventions

– For a domain variable V, let $\mathrm{dom}(V)$, $\min(V)$ and $\max(V)$ respectively denote the set of possible values of V, the smallest possible value of V and finally the largest feasible value of V. The statement $V :: \min..\max$, where $\min$ and $\max$ are two integers such that $\min$ is less than or equal to $\max$, creates a domain variable V for which the initial domain is made up from all values between $\min$ and $\max$ inclusive. Similarly, the statement $V :: v_1, v_2, \ldots, v_l$, where $v_1, v_2, \ldots, v_l$ are distinct integers, creates a domain variable V for which the initial domain is made up from all values $v_1, v_2, \ldots, v_l$. We call *range* of a domain variable V the interval of consecutive values $[\min(V), \max(V)]$. A domain variable for which the possible values consists of one single interval of consecutive values is called an *interval variable*.

– For each possible value v of the *val* attribute of an item of the *Values* collection, let $\mathrm{weight}(v)$ denotes the *weight* attribute associated to the same item.

– We say that a set of values $\mathcal{V}al$ *covers* a set of variables $\mathcal{V}ar$ if the domain of every variable of $\mathcal{V}ar$ intersects $\mathcal{V}al$.

[3] A *complete* filtering algorithm for a given constraint is a filtering algorithm that removes all values that do not occur in at least one solution of the constraint.

2 Pruning According to the Maximum Number of Distinct Values

This section considers an important case of the *sum_of_weights_of_distinct_values* constraint where all the weights are equal to 1 and where one restricts the maximum number of distinct values taken by a set of variables, that is the maximum value of the *Cost* variable. This covers the domination problem explained in Sect. 8. A filtering algorithm for this case was already provided in [2, page 216]. The first part of this section extends this algorithm in order to systematically remove all infeasible values when each assignment variable is an interval variable. This first algorithm is valid, but incomplete, when there are holes in some domain of the assignment variables. Therefore, the second part of this section provides some deduction rules, which allow taking partially into account holes in the domains of the assignment variables. Some of these rules also use the first algorithm, which we now introduce.

2.1 A Complete Filtering Algorithm for Interval Variables

The basic idea of the algorithm for finding a lower bound is to construct a subset of the assignment variables such that no two variables of that subset have a common value in their respective domains. The algorithm is organized in four main steps as follows:

– The first step computes a sharp lower bound of the number of distinct values when all the domains of the assignment variables are intervals,
– The last three steps are only used when the lower bound is equal to the maximum number of distinct values. Their aim is to find all values that, if they were taken by an assignment variable, would lead to use more than $\max(Cost)$ distinct values.

We now explain the details of the four steps:

– Let us denote by $V_1, V_2, \ldots, V_n$ the assignment variables sorted in increasing order of their minimum value. Lines 2-14 of Alg. 1 partition $V_1, V_2, \ldots, V_n$ into `lower_bound`[4] groups of consecutive variables by scanning the variables in order of increasing minimum value and by starting a new group each time `reinit` is set to `TRUE` (see line 7, when `low` is greater than `up`). The different groups of variables can be characterized as follows: The first variable of the first group is V_1, while the first variable of the *i*-th (*i*>1) group is the variable next to the last variable of the *i*−1-th group; the last variable of the last group is V_n, while the last variable of the *i*-th (*i*>1) group, starting at variable V_f, is the variable V_l, such that $l \ (f \le l \le n)$ is the largest integer satisfying the following condition:
$\text{minimum}(\max(V_f), \max(V_{f+1}), \ldots, \max(V_l)) - \text{maximum}(\min(V_f), \min(V_{f+1}), \ldots, \min(V_l)) \ge 0$.
We first justify the fact that `lower_bound` is a lower bound of the number of distinct values: If for each group we consider the variable with the smallest maximum value and the smallest index in case of tie, then we have a total of

[4] `lower_bound` is the value of the lower_bound variable present in Algorithm 1 after finishing the execution of Alg. 1.

`lower_bound` pairwise[5] non-intersecting variables. We now explain why the lower bound is sharp when the domain of each assignment variable consists of one interval. For each group $\mathcal{G}$, consider the smallest maximum value minus one of the variables of $\mathcal{G}$, we have that each interval variable of $\mathcal{G}$ can take this value and therefore we can build an assignment which only uses `lower_bound` distinct values.

- Lines 15-26 of Alg. 1 partition the set of assignment variables $V_1,\ldots,V_n$ by scanning the variables in order of decreasing minimum value and by starting a new group each time `reinit` is set to `TRUE`. For each group of consecutive variables it records in $\texttt{low\_backward}[j]$ $(j \in 1..\texttt{lower\_bound})$ the largest minimum value of the variables of the group.
- Lines 27-34 of Alg. 1 compute the intervals $\texttt{kinf}[j],\ldots,\texttt{ksup}[j]$ $(j \in 1..\texttt{lower\_bound})$ of consecutive values that are feasible for the assignment variables. These intervals are calculated as follows:
  - $\texttt{kinf}[j]$ is the largest minimum value of the variables of the j-th $(j \in 1..\texttt{lower\_bound}-1)$ group of variables constructed during the first step for which the largest value is strictly less than $\texttt{low\_backward}[j+1]$. For $j = \texttt{lower\_bound}$, $\texttt{kinf}[j]$ contains the largest minimum value of the variables of the *`lower_bound`*-th group of variables.
  - $\texttt{ksup}[j]$ is the smallest maximum value of the variables of the j-th group of variables constructed during the first step.
- Lines 35-39 of Alg. 1 remove from the assignment variables those values that do not belong to the intervals computed at the previous step.

We now show the correctness of the pruning. On one side, taking any value of one of the feasible intervals allows to construct one complete assignment for variables $V_1,\ldots,V_n$ such that we use `lower_bound` distinct values. On the other side, consider a value v that does not belong to one of the intervals. We show that fixing any variable $V_1,\ldots,V_n$ to v leads to using at least $\texttt{lower\_bound}+1$ distinct values. This comes from the following observations. First note that, for covering all variables that have a maximum value less than or equal to $\texttt{ksup}[j]$ $(j \in 1..\texttt{lower\_bound})$, we need at least j distinct values. Second observe that, for covering all variables that have a minimum value greater than or equal to $\texttt{kinf}[j]$ $(j \in 1..\texttt{lower\_bound})$, we need at least $\texttt{lower\_bound} - j + 1$ distinct values. Since the two sets of variables do not intersect, it follows that, if we take a value v such that $\texttt{ksup}[j] < v < \texttt{kinf}[j+1]$ $(j \in 1..\texttt{lower\_bound}-1)$, we will need at least $\texttt{lower\_bound}+1$ distinct values for covering all the assignment variables.

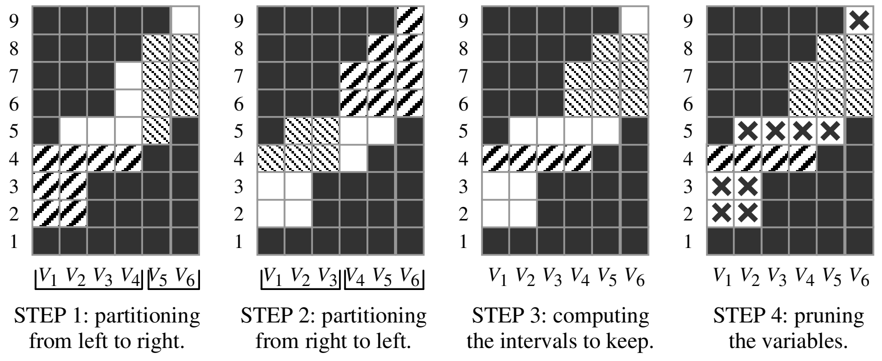
Note that, besides the initial sorting phase and the final pruning, all the other parts of Alg. 1 are in $O(n)$. Thus the overall complexity of Alg. 1 is $O(n \cdot \log n + n \cdot p)$,

[5] Two domain variables are called *non-intersecting variables* when they don't have any value in common.

where p is the number of values to remove. We will now illustrate the different steps of Alg. 1 on the following example.

```
1. V1::2..4  V2::2..5  V3::4..5  V4::4..7  V5::5..8  V6::6..9  Cost::0..2
2. sum_of_weights_of_distinct_values({var-V1,var-V2,var-V3,var-V4,var-V5,var-V6},
3.                              {val-1 weight-1, val-2 weight-1, val-3 weight-1,
4.                               val-4 weight-1, val-5 weight-1, val-6 weight-1,
5.                               val-7 weight-1, val-8 weight-1, val-9 weight-1},Cost)
```

Example 1. Instance used for illustrating the different steps of Alg. 1.



9
8
7
6
5
4
3
2
1
V1 V2 V3 V4 V5 V6
STEP 1: partitioning from left to right.
STEP 2: partitioning from right to left.
STEP 3: computing the intervals to keep.
STEP 4: pruning the variables.

Fig. 1. Illustration of the different steps of Alg. 1 on Example 1

In the four pictures of Fig. 1, each assignment variable V_1,V_2,V_3,V_4,V_5,V_6 corresponds to a given column and each value to a row. Values that do not belong to the domain of a variable are put in black. We now explain each step:

- Step 1 computes a lower bound of the number of distinct values by scanning the variables V_1,V_2,V_3,V_4,V_5,V_6. It builds two groups of adjacent variables V_1,V_2,V_3,V_4 and V_5,V_6. The intervals $[low,up]$ (see lines 5-6 of Alg.1) computed as we scan the variables are dashed. For instance after considering variable V_3 we get the interval $[4,4]$.
- Step 2 scans V_1,V_2,V_3,V_4,V_5,V_6 from right to left in order to initialize the `low_backward` array. After finishing the first group of variables V_6,V_5,V_4 it set `low_backward`$[2]$ to $\text{maximum}(\min(V_6),\min(V_5),\min(V_4))=6$. Finally, after finishing the last group of variables, V_3,V_2,V_1 it set `low_backward`$[1]$ to $\text{maximum}(\min(V_3),\min(V_2),\min(V_1))=4$. Like in the previous step, the intervals $[low,up]$ computed as we scan the variables are dashed.
- Step 3 scans V_1,V_2,V_3,V_4,V_5,V_6 from left to right and computes the intervals of values to keep in order to not exceed two distinct values. The lower bound $\text{kinf}[1]$ of the first interval is obtained by first selecting within the variables of the first group (i.e. V_1,V_2,V_3,V_4) those variables for which the maximum value is strictly less than `low_backward`$[2]=6$. Then we take the maximum of the smallest value of the variables we just select (i.e. V_1,V_2,V_3), which is 4. The upper bound $\text{ksup}[1]$ of the first interval is the minimum value of the largest value of the variables of the first group, namely 4. In a similar way we obtain that $\text{kinf}[2]=6$ and $\text{ksup}[2]=8$. On the corresponding picture, the intervals of values to keep are dashed.

- Step 4 removes all values that are not located within one of the intervals of values to keep. These values to remove are marked with a cross.

```
   PARTITION THE VARIABLES OF V[1..n] IN GROUPS OF CONSECUTIVES VARIABLES
 1 Sort V[1..n] in increasing minimum value;
 2 reinit:=TRUE; i:=1; lower_bound:=1; start_prev_group:=1;
 3 WHILE (reinit AND i≤n) OR ((NOT reinit) AND i<n) DO
 4   IF NOT reinit THEN i:=i+1;
 5   IF reinit OR low<min(V[i])[6] THEN low:=min(V[i]);
 6   IF reinit OR up >max(V[i]) THEN up :=max(V[i]);
 7   reinit:=(low>up);
 8   IF reinit OR i=n THEN
 9     IF reinit THEN end_prev_group:=i-1 ELSE end_prev_group:=i;
10     start_group[lower_bound]:=start_prev_group;
11       end_group[lower_bound]:=  end_prev_group;
12     start_prev_group:=i;
13   IF reinit THEN lower_bound:=lower_bound+1;
14 adjust minimum value of Cost to lower_bound;
15 IF lower_bound=max(Cost) THEN
     BUILD THE "RIGHTMOST" GROUPS OF VARIABLES
16   reinit:=TRUE; i:=n; j:=lower_bound;
17   WHILE (reinit AND i≥1) OR ((NOT reinit) AND i>1) DO
18     low_before:=low;
19     IF (NOT reinit) THEN i:=i-1;
20     IF reinit OR low<min(V[i]) THEN low:=min(V[i]);
21     IF reinit OR up >max(V[i]) THEN up :=max(V[i]);
22     reinit:=(low>up);
23     IF reinit OR i=1 THEN
24       IF NOT reinit THEN low_before:=low;
25       low_backward[j]:=low_before;
26       IF reinit THEN j:=j-1;
     COMPUTE INTERVALS OF CONSECUTIVE VALUES TO KEEP
27   FOR j:=1 TO lower_bound DO
28     first_kinf:=TRUE; first_ksup:=TRUE;
29     FOR i=start_group[j] TO end_group[j] DO
30       IF  (j=lower_bound OR max(V[i])<low_backward[j+1])
31       AND (first_kinf OR min(V[i])>kinf[j]) THEN
32         kinf[j]:=min(V[i]); first_kinf:=FALSE;
33       IF first_ksup OR max(V[i])<ksup[j] THEN
34         ksup[j]:=max(V[i]); first_ksup:=FALSE;
     REMOVE ALL VALUES WHICH ARE NOT SITUATED WITHIN kinf[j]..ksup[j]
35   FOR i:=1 TO n DO
36     adjust minimum and maximum of V[i] to kinf[1] and ksup[lower_bound];
37   FOR j:=1 TO lower_bound-1 DO
38     IF ksup[j]+1≤kinf[j+1]-1 THEN
39       FOR i:=1 TO n DO remove ksup[j]+1.. kinf[j+1]-1 from V[i];
```

Algorithm 1: A complete filtering algorithm when the weights are 1 and each domain is an interval

2.2 Taking Holes into Account

This section provides deduction rules, which take advantage of the fact that some assignments variables are not interval variables. Within Sect. 2.2, *lower_bound* refers to the lower bound computed by step 1 of Alg. 1.

Unification of Assignment Variables. When the lower bound computed by Alg. 1 is equal to maximum number of possible distinct values (see line 14 of Alg. 1), we have

[6] Throughout the algorithms of this article, the evaluation of boolean expressions is performed from left to right in a lazy way. This explains why `low` does not need to be initialized.

that the assignment variables should take exactly one value within each interval `[kinf[j],ksup[j]]` ($1 \le j \le lower\_bound$). Consequently, if the domain of an assignment variable Var is contained within one of the intervals, all values of the interval that do not belong to the domain of Var should be removed from the domain of all the assignment variables. As a special case of the previous deduction rule, we have that two variables for which the domain is included within the same interval `[kinf[j],ksup[j]]` ($1 \le j \le lower\_bound$) should be unified. Using unification in this case has the following advantages. First, we can forget about one of the variables. Second, we don't need to maintain the consistency between the domains of the variables, which were unified.

Consider the assignment variables of Example 1 after pruning (see step 4 of Fig. 1). Since, both V_5 and V_6 can only take values within interval `[kinf[2],ksup[2]]=[6,8]`, we have that $V_5 = V_6$. Now, assume that value 7 does not belong to dom(V_5). Then it should also be removed from the domain of V_6.

Pruning According to the Value Profile. This paragraph provides a lower bound for the minimum number of distinct values, which takes into account holes in the domains of the assignment variables. It then shows how to prune the domains of the assignment variables according to this bound. The method is based on a profile of number of occurrences of values.

The *profile of number of occurrences of values* gives, for each potential value v of an assignment variable, the number of assignment variables for which the domain contains v. The *profile sequence* $(o_1, o_2, \ldots, o_{|Values|})$ with $o_i \ge o_{i+1}$ corresponds to the different number of assignment variables taking a specific value sorted in decreasing order. We are now in position to define a new lower bound. Throughout this paragraph we use the following example for illustrating the different deduction rules.

```
V1::1,2,4  V2::3,5  V3::4,6  V4::1,3,5  V5::3,6  Cost::0..2
sum_of_weights_of_distinct_values({var-V1,var-V2,var-V3,var-V4,var-V5},
                            {val-1 weight-1, val-2 weight-1, val-3 weight-1,
                             val-4 weight-1, val-5 weight-1, val-6 weight-1},Cost)
```

Example 2. Instance used for illustrating the pruning according to the profile of number of occurrences of values.

Proposition 1
If the n *assignment variables of the* $sum\_of\_weights\_of\_distinct\_values$ *constraint have the profile sequence* $(o_1, o_2, \ldots, o_{|Values|})$ *with* $o_i \ge o_{i+1}$*, then the minimum number of distinct values is greater than or equal to* $\min\{k : (o_1 + o_2 + \cdots + o_k) \ge n\}$.

We now give two rules that prune the assignment variables according to the value profile. The first rule enforces, under certain conditions, to use the value that occurs in most variables, while the second rule removes those values that do not occur in too many variables.

Rule 1: Consider a set of assignment variables of the $sum\_of\_weights\_of\_distinct\_values$ constraint with the profile sequence $(o_1, o_2, \ldots, o_{|Values|})$ with $o_i \ge o_{i+1}$. Let v_1 denotes the value associated to the number of occurrence o_1.

If $\sum_{1 \le i \le \max(Cost)} o_i = n$ and if $o_1 > o_2$ then all assignment variables Var such that $v_1 \in \mathrm{dom}(Var)$ should be fixed to value v_1.

Rule 2: Consider a set of assignment variables of the *sum_of_weights_of_distinct_values* constraint with the profile sequence $(o_1, o_2, \ldots, o_{|Values|})$ with $o_i \ge o_{i+1}$ and let k be the smallest integer such that $(o_1 + o_2 + \cdots + o_k) \ge n$. Let v_i $(1 \le i \le |Values|)$ denote the value associated to the number of occurrences o_i. We can remove a value v_j $(k < j \le |Values|)$ from all the assignment variables if $\sum_{1 \le i \le \max(Cost)} o_i - (o_k - o_j) < n$.

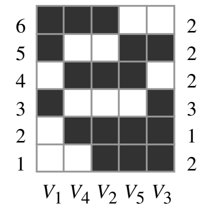


Fig. 2. Profile of number of occurrences of values

Let us illustrate the computation of the lower bound as well as the use of the deduction rules on Example 2. In Fig. 2, each assignment variable V_1, V_2, V_3, V_4, V_5 corresponds to a given column and each value to a row. Values that do not belong to the domain of a variable are put in black. For each possible value the corresponding rightmost integer gives the number of assignment variables that can effectively take this value. The associated profile sequence $(o_1, o_2, o_3, o_4, o_5, o_6)$ is equal to $(3,2,2,2,2,1)$ and the corresponding values $v_1, v_2, v_3, v_4, v_5, v_6$ are respectively equal to $3,1,4,5,6,2$. Since the smallest value k such that $(o_1 + o_2 + \cdots + o_k) \ge n = 5$ is equal to 2, we need at least two distinct values for covering all assignment variables V_1, V_2, V_3, V_4, V_5. Let us now consider Rule 1: Since both $\sum_{1 \le i \le \max(Cost)} o_i = \sum_{1 \le i \le 2} o_i = 3 + 2 = 5 = n$ and $o_1 > o_2$ hold, we apply Rule 1 and therefore fix V_2, V_4 and V_5 to value $v_1 = 3$. Finally, consider Rule 2: Since $\sum_{1 \le i \le \max(Cost)} o_i - (o_k - o_6) = \sum_{1 \le i \le 2} o_i - (o_2 - o_6) = 5 - (2 - 1) < n = 5$, we remove value $v_6 = 2$ from V_1.

3 Lower Bound for the Sum of the Weights of the Distinct Values

This section presents an $O(n \cdot \log n + m)$ algorithm for computing a lower bound for the sum of the weights of the distinct values. When the domain of each assignment variable is an interval this lower bound is tight.

Principle of the Algorithm

The algorithm for computing a lower bound consists of the following steps:

– We first select (see steps A, B, C) from the assignment variables $Var_1, Var_2, \ldots, Var_n$ of the *sum_of_weights_of_distinct_values* constraint a subset of variables $\mathcal{V}ar$ for which the following property holds: If the domain of every assignment variable is an interval, then any set of values covering all variables of $\mathcal{V}ar$ allows also to cover all variables of $Var_1, Var_2, \ldots, Var_n$. The construction of $\mathcal{V}ar$ is based on the following two observations. Firstly, if $\mathrm{dom}(Var_i)$ $(1 \le i \le n)$ is included in or equal to $\mathrm{dom}(Var_j)$ $(j \ne i,\ 1 \le j \le n)$ then the sum of the weights of the distinct values does not change if Var_j is not considered. Secondly, we try to obtain a set $\mathcal{V}ar$ with a specific property[7] for which we have a polynomial algorithm which finds a tight lower bound.

– We then compute (see step D) a lower bound for covering all variables of $\mathcal{V}ar$. This lower bound will be tight when the domain of each variable of $\mathcal{V}ar$ is an interval.

Throughout sections 3 and 4, we illustrate the different phases of the algorithm on the instance given in the following example. Lines 1 and 2 of Example 3 declare the minimum and maximum value for each assignment variable as well as for the cost variable. Lines 3 to 9 state a *sum_of_weights_of_distinct_values* constraint where we have 14 assignment variables (see lines 3-4) and their 17 potential values (see lines 5-9).

```
1. V1::0..6 V2::1..7  V3:: 1..11 V4:: 2..10 V5:: 2.. 7 V6:: 3.. 8 V7::5..11 V8::5..8
2. V9::6..9 Va::6..12 Vb::11..12 Vc::11..13 Vd::13..15 Ve::14..16 Cost::0..18
3. sum_of_weights_of_distinct_values({var-V1,var-V2,var-V3,var-V4,var-V5,var-V6,var-V7,
4.                                    var-V8,var-V9,var-Va,var-Vb,var-Vc,var-Vd,var-Ve},
5. {val-0  weight-7 , val-1  weight-12, val-2  weight-3, val-3  weight-10,
6.  val-4  weight-6 , val-5  weight-6 , val-6  weight-9, val-7  weight-5 ,
7.  val-8  weight-10, val-9  weight-1 , val-10 weight-7, val-11 weight-1 ,
8.  val-12 weight-5 , val-13 weight-8 , val-14 weight-9, val-15 weight-10,
9.  val-16 weight-4},Cost)
```

Example 3. Instance used for illustrating Alg. 2 and Alg. 3.

Fig. 3 will be used at each step of the algorithm to depict specific information. On that figure, each assignment variable $V_1, V_2, \ldots, V_e$ of Example 3 corresponds to a given column and each value to a row. Values that do not belong to the domain of a variable are put in black. Further explanations about Fig. 3 will come as we develop the different steps of Alg. 2 and 3.

A. Sorting the Assignment Variables. We first sort the assignment variables $Var_1, Var_2, \ldots, Var_n$ in increasing order of their minimum value (see line 2 of Alg. 2), which takes $O(n \cdot \log n)$. These sorted variables will be denoted by $V_1, V_2, \ldots, V_n$ throughout the rest of this section and of the next section.

[7] The property is given by Condition (1) of Proposition 2.

B. Making a First Selection of Variables to Cover. Let us first introduce the notion of *stair*, which is needed at this stage. A stair is a set of consecutive variables $V_i, V_{i+1}, \ldots, V_j$ $(i \le j)$ of $V_1, V_2, \ldots, V_n$ such that all the following conditions hold:

$$\min(V_i) = \ldots = \min(V_j), \quad i = 1 \text{ or } \min(V_{i-1}) \neq \min(V_i), \quad j = n \text{ or } \min(V_{j+1}) \neq \min(V_j).$$

Part (B) of Fig. 3 indicates the stairs of $V_1, V_2, \ldots, V_e$. We have the following nine stairs $\{V_1\}, \{V_2, V_3\}, \{V_4, V_5\}, \{V_6\}, \{V_7, V_8\}, \{V_9, V_a\}, \{V_b, V_c\}, \{V_d\}$ and $\{V_e\}$ which respectively correspond to the variables which have value 0, 1, 2, 3, 5, 6, 11, 13 and 14 as a minimum value.

Lines 3-7 of Alg. 2 select for each stair the leftmost variable with the smallest maximum value. The selected variable is called the *representative* of the stair. We scan the variables once and therefore this phase takes $O(n)$.

If the domains of all the variables of a stair contain the domain of its representative, then covering the representative allows also to cover all variables of that stair.
Part (C) of Fig. 3 indicates for each stair its representative. For instance the representatives of the first six stairs $\{V_1\}, \{V_2, V_3\}, \{V_4, V_5\}, \{V_6\}, \{V_7, V_8\}$ and $\{V_9, V_a\}$ are respectively V_1, V_2, V_5, V_6, V_8 and V_9.

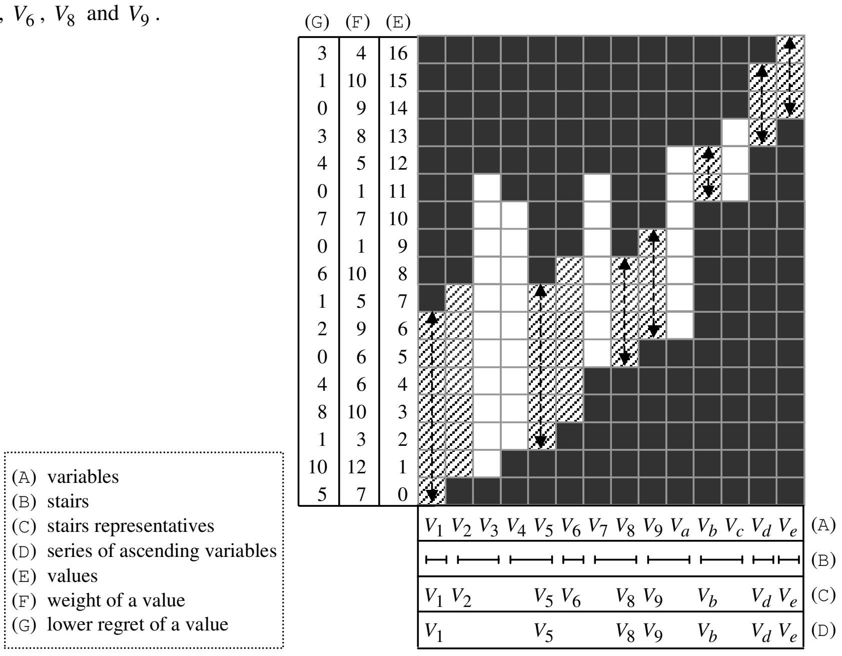


Fig. 3. Computing the overall lower bound and the lower regret of each value

C. Restricting Further the Set of Variables to Cover. The goal of this step is to further restrict the set of representatives $\mathcal{Representatives}$ computed at the previous step to a subset $\mathcal{Ascending}$ so that both of the following properties hold:

– The range of a newly eliminated variable (i.e. a variable belonging to $\mathcal{Representatives} - \mathcal{Ascending}$) contains the range of at least one variable of $\mathcal{Ascending}$.
– There does not exist two distinct variables V_a, V_b of $\mathcal{Ascending}$ for which both $\min(V_b) \leq \min(V_a)$ and $\max(V_a) \leq \max(V_b)$ hold.

Under the assumption that each domain consists of one single interval, the first property guarantees that covering all variables of $\mathcal{Ascending}$ allows also covering all eliminated variables of $\mathcal{Representatives} - \mathcal{Ascending}$ without using any extra value, therefore without any extra cost. The second property comes from the fact that, as we will explain in the next paragraph, we know how to compute a tight lower bound for such a configuration of variables whose domains are all intervals.

Selecting a series of ascending variables $\mathcal{Ascending}$ is achieved as follows. We scan back from right to left the variables of $\mathcal{Representatives}$ (see lines 8-10 of Alg. 2). During this scan, we mark a variable (see instruction “`stair[s]:=-1`” at line 10 of Alg. 2) of $\mathcal{Representatives}$ if its maximum value is greater than or equal to the smallest maximum value encountered so far, where the initial maximum value is set to a value strictly greater than the maximum value of the representative of the last stair (see line 8 of Alg. 2). Finally, we compress all unmarked variables so that their indices are put in adjacent entries of the `stair` array. Since these phases require scanning the variables twice, their complexity is $O(n)$.

Part (D) of Fig. 3 shows the series of ascending variables $\mathcal{Ascending}$ extracted from $\mathcal{Representatives}$. It is obtained as follows: From the representatives V_1, V_2, V_5, V_6, V_8, V_9, V_b, V_d, V_e of the stairs, we first eliminate V_6 since its maximum value is greater than or equal to the maximum value of V_8. We also eliminate V_2 since its maximum value is greater than or equal to the maximum value of V_5. We finally get the series of ascending variables V_1, V_5, V_8, V_9, V_b, V_d, V_e which have the following strictly increasing minimum and maximum values: 0,2,5,6,11,13,14 and 6,7,8,9,12,15,16. On Fig. 3, a dashed arrow depicts those variables that belong to the series of ascending variables.

D. Computing a Lower Bound for a Series of Ascending Variables. We now come to the point were we explain how to compute a lower bound for the series of ascending variables $\mathcal{Ascending}$. Since we assume that the domain of each variable is an interval, the goal is to find out a subset of distinct values $v_1, v_2, \ldots, v_k$ intersecting $[\min(V), \max(V)]$ for each variable $V \in \mathcal{Ascending}$. In addition, we want to minimize the quantity $\sum_{i=1}^{k} \text{weight}(v_i)$. For this purpose, we use the following proposition.

Proposition 2
Assume that, for each possible value v of a domain variable V_p, we know a tight lower bound $tlb_{1,2,\ldots,p}^{v}$ for covering all variables $V_1, V_2, \ldots, V_p$ according to the hypothesis that we use value v (0 is a tight lower bound for the empty set).

Furthermore, let V_{p+1} be a domain variable such that

$$\left(\mathrm{dom}\left(V_{p+1}\right)-\mathrm{dom}\left(V_p\right)\right) \;\cap\; \bigcup_{i=1}^{p-1}\mathrm{dom}\left(V_i\right) \;=\; \varnothing\,, \qquad (1)$$

(i.e. all possible values of V_{p+1} that do not belong to $\mathrm{dom}\left(V_p\right)$ do also not belong to the domains of $V_1,V_2,\ldots,V_{p-1}$). For each possible value w of $\mathrm{dom}\left(V_{p+1}\right)$ the following formulas compute a tight lower bound $tlb^{w}_{1,2,\ldots,p+1}$ for covering $V_1,V_2,\ldots,V_{p+1}$ under the hypothesis that we use value w:

– *If* $w \in \mathrm{dom}\left(V_p\right)$ *then* $tlb^{w}_{1,2,\ldots,p+1} = tlb^{w}_{1,2,\ldots,p}$. (2)

– *If* $w \notin \mathrm{dom}\left(V_p\right)$ *then* $tlb^{w}_{1,2,\ldots,p+1} = \min\limits_{v\in \mathrm{dom}\left(V_p\right)}\left(tlb^{v}_{1,2,\ldots,p}\right) + \mathrm{weight}(w)$. (3)

Proof of Proposition 2

(2) arises from the fact that, if $w \in \mathrm{dom}\left(V_p\right)$, we don't need any extra value for covering the new variable V_{p+1}. Finally, (3) originates from the fact that, using a value w, which for sure does not belong to the domains of $V_1,V_2,\ldots,V_p$, will not allow covering any variables of $V_1,V_2,\ldots,V_p$. Therefore, we can add to the weight of w the smallest tight lower bound for covering $V_1,V_2,\ldots,V_p$ associated to the different possible values v of the domain of V_p. □

Lines 12 to 20 of Alg. 2 use Proposition 2 in order to compute a lower bound for the sum of the weights of the distinct values. Line 13 iterates through the variables of $\mathcal{A}scending$. Moreover, in order to satisfy Condition (1), the variables are considered in increasing order of their minimum value and we relax the fact that they may not consist of one single interval. This relaxation is achieved by lowering the quantity $\min\limits_{v\in \mathrm{dom}\left(V_p\right)}\left(tlb^{v}_{1,2,\ldots,p}\right)$ of (3) to $\min\limits_{\min\left(V_p\right)\le v\le \max\left(V_p\right)}\left(tlb^{v}_{1,2,\ldots,p}\right)$. This last quantity will be denoted by $tlb_{1,2,\ldots,p}$. In order to keep the overall complexity of lines 12 to 20 to $O(m)$, we use a sliding window for computing the quantity $\min\limits_{\min\left(V_p\right)\le v\le \max\left(V_p\right)}\left(tlb^{v}_{1,2,\ldots,p}\right)$ without rescanning the values between $\min\left(V_p\right)$ and $\max\left(V_p\right)$. Those values that belong to the range of the current variable V_p are kept in this sliding window for which we now describe the contents.

For a series of strictly increasing values $v_{first}, v_{first+1},\ldots,v_{last}$ ($1 \le first \le last \le m$), let $\mathrm{weight}\left(v_{last}\right)$ be the Δ-th smallest distinct value[8] among the values of $\{\mathrm{weight}\left(v_{first}\right), \mathrm{weight}\left(v_{first+1}\right),\ldots,\mathrm{weight}\left(v_{last}\right)\}$. The sliding window records the following information:

– `key`$[low+i]$ $(0 \le i < \Delta)$ contains the $(i+1)$-th smallest distinct value within $\{\mathrm{weight}\left(v_{first}\right),\ \mathrm{weight}\left(v_{first+1}\right),\ldots,\mathrm{weight}\left(v_{last}\right)\}$,

[8] For instance, the second smallest distinct value of values 9,4,1,3,1,4 is equal to 3.

– pos$[low+i]$ $(0 \le i < \Delta)$ holds the largest value $v_k\,(first \le k \le last)$ such that weight$(v_k)=$key$[low+i]$.

Consider the possible values 0,1,2,3,4,5,6 of the first variable V_1 as well as their corresponding weights 7,12,3,10,6,6,9. Since 9 is the third smallest distinct value of the strictly increasing sequence 3,6,9 extracted from the weights, the first three entries of the key and pos arrays of the sliding window associated to values 0,1,2,3,4,5,6 are initialized as follows: key[1]=3, pos[1]=3, key[2]=6, pos[2]=6, key[3]=9, pos[3]=7.

```
PURPOSE  Compute a lower bound for the sum of the weight of the distinct values taken
         by Var[1],Var[2],..,Var[n] and update the minimum value of the Cost variable.
INPUT
• n           : Total number of variables.
• m           : Total number of values.
• Var   [1..n]: The variables.
• Value [1..m]: Contains the values of the val attributes of the second argument
                of the sum_of_weights_of_distinct_values constraint sorted in
                increasing order.
• Weight[1..m]: Weight associated to the different values.
• Cost        : The cost variable.
   INITIALIZATION
 1 nstair:=0; FOR i:=1 TO n DO V[i]:=Var[i];
 2 Sort V[1..n] in increasing order of minimum value of V[1..n];
   SELECT A FIRST SUBSET OF VARIABLES
 3 FOR i:=1 TO n DO
 4 | new_stair:=(nstair=0 OR level_stair<min(V[i]));
 5 | IF new_stair THEN nstair:=nstair+1; level_stair:=min(V[i]);
 6 | IF new_stair OR smallest_max_stair>max(V[i]) THEN
 7 | | stair[nstair]:=i; smallest_max_stair:=max(V[i]);
   RESTRICT FURTHER THE FIRST SUBSET OF VARIABLES
 8 cut:=max(V[stair[nstair]])+1;
 9 FOR s:=nstair TO 1 (STEP -1) DO
10 | IF max(V[stair[s]])≥cut THEN stair[s]:=-1 ELSE cut:=max(V[stair[s]]);
11 r:=0; FOR s:=1 TO nstair DO   IF stair[s]≠-1 THEN r:=r+1; stair[r]:=stair[s];
   COMPUTE LOWER BOUND FOR COVERING THE SELECTED SUBSET OF VARIABLES
12 low:=1; up:=0; lower_bound:=0;
13 FOR s:=1 TO r DO
14 | minv:=min(V[stair[s]]); maxv:=max(V[stair[s]]);
15 | WHILE low≤up AND pos[low]<minv DO low:=low+1;
16 | IF s>1 AND minv<max(V[stair[s-1]])+1 THEN minv:=max(V[stair[s-1]])+1;
17 | FOR v:=minv TO maxv DO
18 | | WHILE low≤up AND key[up]≥Weight[v-Value[1]+1]+lower_bound DO up:=up-1;
19 | | up:=up+1; pos[up]:=v; key[up]:=Weight[v-Value[1]+1]+lower_bound;
20 | lower_bound:=key[low];
   ADJUST LOWER BOUND OF COST
21 adjust minimum of Cost to lower_bound;
```

Algorithm 2: Lower bound for the sum of the weights of the distinct values

This sliding window moves at each step of the iteration when we process the next variable V_{k+1}. Those values that are smaller than the minimum value of V_{k+1} leave the sliding window (see line 15 of Alg. 2), while those values that belong to $\max(\max(V_k)+1,\min(V_{k+1})),\max(\max(V_k)+1,\min(V_{k+1}))+1,..,\max(V_{k+1})$ enter the sliding window (see lines 18-19 of Alg. 2). Accessing the leftmost position of the key array of the sliding window retrieves the minimum weight without any scanning (see line 20 of Alg. 2). Each move of the sliding window (i.e. removing or adding a value) is achieved by shrinking the leftmost or the rightmost parts of the sliding window and by possibly extending the sliding window by one position to the right. The key point is that each value is inserted and removed at most once from the sliding window and that each scan over a value removes this value. Since we have no more than m elements, the overall complexity for updating the sliding window is $O(m)$.

We now give the detail of the first steps of the computation of the lower bound on the series of ascending variables V_1, V_5, V_8, V_9, V_b, V_d, V_e.

- At the first iteration we compute $tlb_1^0 = \text{weight}(0) = 7$, $tlb_1^1 = \text{weight}(1) = 12$, $tlb_1^2 = \text{weight}(2) = 3$, $tlb_1^3 = \text{weight}(3) = 10$, $tlb_1^4 = \text{weight}(4) = 6$, $tlb_1^5 = \text{weight}(5) = 6$, $tlb_1^6 = \text{weight}(6) = 9$. The lower bound tlb_1 for covering V_1 is equal to $\min\left(tlb_1^0, tlb_1^1, tlb_1^2, tlb_1^3, tlb_1^4, tlb_1^5, tlb_1^6\right) = 3$.
- At the second iteration we have that $tlb_{1,2}^i = tlb_1^i$ ($2 \le i \le 6$) and we compute $tlb_{1,2}^7 = tlb_1 + \text{weight}(7) = 3 + 5 = 8$. The lower bound tlb_2 for covering V_1, V_5 is equal to $\min\left(tlb_{1,2}^2, tlb_{1,2}^3, tlb_{1,2}^4, tlb_{1,2}^5, tlb_{1,2}^6, tlb_{1,2}^7\right) = 3$.
- At the third iteration we have that $tlb_{1,2,3}^i = tlb_{1,2}^i$ ($5 \le i \le 7$) and we compute $tlb_{1,2,3}^8 = tlb_2 + \text{weight}(8) = 3 + 10 = 13$. The lower bound tlb_3 for covering V_1, V_5, V_8 is equal to $\min\left(tlb_{1,2,3}^5, tlb_{1,2,3}^6, tlb_{1,2,3}^7, tlb_{1,2,3}^8\right) = 6$.

Finally after iterating through the remaining variables V_9, V_b, V_d, V_e we get an overall lower bound of 17.

Taking into account the complexity of all the intermediate steps described above leads to an overall complexity of $O(n \cdot \log n + m)$ for computing a lower bound for the sum of the weights of the distinct values.

4 Pruning According to the Lower Regret

This section first introduces the notion of *lower regret*, denoted $\underline{\text{regret}}(Var, v)$, associated to a pair Var, v where Var is an assignment variable and v a value. Then it extends the algorithm of the previous section in order to prune the assignment variables according to this lower regret and to the maximum possible value of the $Cost$ variable. Finally it indicates how to derive a potentially better lower bound for the sum of the weights of the distinct values from the lower regret as well as from the potential holes in the assignment variables.

Lower Regret of a Value. The lower regret associated to a pair Var, v, where Var is an assignment variable and v a value, is the minimum increase of the lower bound of the sum of the weights of the distinct values under the hypothesis that Var is assigned to value v.

When we compute the lower bound, we want to minimize the sum of the weights associated to the values used by at least one variable. So, as soon as a variable takes a given value v, all variables that can take that value, should be assigned to v since this does not imply any additional cost. It follows that for such variables $\underline{\text{regret}}(Var_i, v) = \underline{\text{regret}}(Var_j, v)$ ($1 \le i, j \le n$). Having this remark in mind, we now show how to compute the lower regret associated to a value v (denoted $\underline{\text{regret}}(v)$), no matter which variable takes this value.

Proposition 3
The lower bound on the sum of the weights of the distinct values under the hypothesis that a variable is assigned to value v is equal to

$$tlb_{1,2,\ldots,f} + \text{weight}(v) + tlb_{l,l+1,\ldots,n} \ , \tag{4}$$

where f is largest index of the variables of $\mathcal{A}scending$ such that $\max(V_f) < v$ [9], and l is the smallest index of the variables of $\mathcal{A}scending$ such that $\min(V_l) > v$ [7]. The lower regret of value v is equal to

$$tlb_{1,2,\ldots,f} + \text{weight}(v) + tlb_{l,l+1,\ldots,n} - tlb_{1,2,\ldots,n} \ . \tag{5}$$

Proof of Proposition 3
Since we have to use value v we cover with value v all variables V_i ($1 \le i \le n$) such that $\min(V_i) \le v \le \max(V_i)$ with a cost of $\text{weight}(v)$. In addition, we have also to cover all variables for which the maximum value is strictly less than v as well as all variables for which the minimum value is strictly greater than v. For evaluating the two costs, we use Proposition 2 and the fact that both the minimum and the maximum values of the series of ascending variables are strictly increasing. This leads to (4). The lower regret is obtained by subtracting from (4) the lower bound computed in the previous section. □

Extending Alg. 2 for Computing the Lower Regret. Alg. 3 shows how to extend Alg. 2 in order to compute the lower regret of each value. It also explains how to prune the assignment variables according to the lower regret and to the maximum allowed cost. Let r denote the number of elements of $\mathcal{A}scending$, namely the number of selected variables to cover. An array `lower_regret[1..`m`]` records the lower regret of each value which is computed as described below:

- Line 20 of Alg. 2 computes the quantity $tlb_{1,2,\ldots,f}$ ($1 \le f \le r$) present in (5). Since we need this quantity for evaluating the lower regret, we record it at entry f of the array `sbefore[0..`r`]` (`sbefore[0]` is initialized to 0 and corresponds to $tlb_{\varnothing}$).
- Since we also need to compute quantity $tlb_{l,l+1,\ldots,n}$ ($1 \le l \le r$) we reuse a similar algorithm as in lines 12-20 of Alg. 2 where we now scan the variables by decreasing indices. We record this quantity at entry $n - f + 1$ of the array `safter[0..`r`]` (`safter[0]` is initialized to 0 and corresponds to $tlb_{\varnothing}$).
- For each value v we need to compute the largest index of the non-covered variables of $\mathcal{A}scending$ (see index f in (4)). This is done with a complexity of

[9] If no such value exists, $tlb_{\varnothing}$ is equal to 0.

$O(m)$ by scanning all the values at lines 31-34 of Alg. 3 and by storing this information at entry $v-\texttt{Value}[1]+1$ of the array `first[1..m]`.

- For each value v we also need to compute the smallest index of the non-covered variables of $\mathcal{A}scending$ (see index l in (4)). This is also done with a complexity of $O(m)$ by scanning all the values at lines 35-39 of Alg. 3 and by storing this information at entry $v-\texttt{Value}[1]+1$ of the array `last[1..m]`.
- Finally, in a last phase (see lines 39-42 of Alg. 3), we compute the lower regret of each value (see line 40) and remove (see line 42) from all assignment variables those values for which the sum of the lower bound and the corresponding lower regret exceeds (see line 41) the maximum allowed cost.

At the end of line 12 of Alg. 2 we add the following instruction:

```
12 wbefore[0]:=0;
```

At the end of line 20 of Alg. 2 we add the following instruction:

```
20 wbefore[s]:=lower_bound;
```

After line 21 of Alg. 2 we add the following lines:

```
   COMPUTE LOWER BOUND FOR COVERING THE SELECTED SUBSET OF VARIABLES
22 low:=1; up:=0; lower_bound:=0; wafter[0]:=0;
23 FOR s:=r TO 1 (STEP -1) DO
24   maxv:=max(V[stair[s]]); minv:=min(V[stair[s]]);
25   WHILE low≤up AND pos[low]>maxv DO low:=low+1;
26   IF s<r AND maxv>min(V[stair[s+1]])-1 THEN maxv:=min(V[stair[s+1]])-1;
27   FOR v:=maxv TO minv (STEP -1) DO
28     WHILE low≤up AND key[up]≥Weight[v-Value[1]+1]+lower_bound DO up:=up-1;
29     up:=up+1; pos[up]:=v; key[up]:=Weight[v-Value[1]+1]+lower_bound;
30   lower_bound:=key[low]; wafter[r-s+1]:=lower_bound;

   COMPUTE FIRST NON-COVERED SELECTED VARIABLES BEFORE EACH VALUE
31 FOR v:=Value[1] TO max(V[stair[1]]) DO first[v-Value[1]+1]:=0;
32 FOR s:=1 TO r-1 DO
33   FOR v:=max(V[stair[s]])+1 TO max(V[stair[s+1]]) DO first[v-Value[1]+1]:=s;
34 FOR v:=max(V[stair[r]])+1 TO Value[m] DO first[v-Value[1]+1]:=r;

   COMPUTE FIRST NON-COVERED SELECTED VARIABLE AFTER EACH VALUE
35 FOR v:=Value[m] TO min(V[stair[r]]) (STEP -1) DO last[v-Value[1]+1]:=0;
36 FOR s:=r TO 2 (STEP -1) DO
37   FOR v:=min(V[stair[s]])-1 TO min(V[stair[s-1]]) (STEP -1) DO last[v-Value[1]+1]:=r-s+1;
38 FOR v:=min(V[stair[1]])-1 TO Value[1] (STEP -1) DO last[v-Value[1]+1]:=r;

   PRUNE ACCORDING TO LOWER REGRET AND MAXIMUM COST
39 FOR ival:=1 TO m DO
40   lower_regret[ival]:=wbefore[first[ival]]+Weight[ival]+wafter[last[ival]]-lower_bound;
41   IF lower_bound+lower_regret[ival]>max(Cost) THEN
42     FOR i:=1 TO n DO remove value Value[ival] from V[i];
```

Algorithm 3: Extending Alg. 2 for computing the lower regret and pruning according to it

Computing the lower regret of all values has a complexity of $O(m)$, while pruning according to the lower regret is done in $O(m+q\cdot n)$[10] where q is the number of values for which the condition $lower\_bound + \text{lower\_regret}[ival] > \max(Cost)$ holds. This leads to an overall complexity of $O(n\cdot \log n + m + q\cdot n)$ for one invocation of the filtering algorithm. In order to improve in practice the running time, we store for each value the fact whether or not it was removed from the different assignment variables. Thus for a value that was already removed from the assignment variables we save the iteration through the assignment variables.

[10] We assume that removing a value from a domain variable is done in $O(1)$.

We now give the detail of the computation of the lower regret for the first 9 values 0,1,2,3,4,5,6,7 and 8. We first start by giving the content of the four arrays `sbefore`, `safter`, `first` and `last` initialized by Alg. 3.

`sbefore`	0	3	3	6	7	8	16	17
`safter`	0	4	9	10	11	15	15	17
	0	1	2	3	4	5	6	7

`first`	0	0	0	0	0	0	0	1	2	3	4	4	4	5	5	5	6
`last`	6	6	5	5	5	4	3	3	3	3	3	2	2	1	0	0	0
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

By using line 40 of Alg. 3 and the content of the four arrays we compute the lower regret of 0,1,2,3,4,5,6,7,8 as follows (when we compute the lower regret of a value *v*, the index used in the different arrays corresponds to the entry of the `Value` table such that `Value[`*ival*`]` is equal to *v*).

- $\underline{\text{regret}}(0) = tlb_{\varnothing} + \texttt{Weight}[1] + tlb_{2,3,4,5,6,7} - tlb_{1,2,3,4,5,6,7}$
  $= \texttt{sbefore}[\texttt{first}[1]] + \texttt{Weight}[1] + \texttt{safter}[\texttt{last}[1]] - \texttt{sbefore}[7] = 0+7+15-17 = 5$,
- $\underline{\text{regret}}(1) = tlb_{\varnothing} + \texttt{Weight}[2] + tlb_{2,3,4,5,6,7} - tlb_{1,2,3,4,5,6,7}$
  $= \texttt{sbefore}[\texttt{first}[2]] + \texttt{Weight}[2] + \texttt{safter}[\texttt{last}[2]] - \texttt{sbefore}[7] = 0+12+15-17 = 10$,
- $\underline{\text{regret}}(2) = tlb_{\varnothing} + \texttt{Weight}[3] + tlb_{3,4,5,6,7} - tlb_{1,2,3,4,5,6,7}$
  $= \texttt{sbefore}[\texttt{first}[3]] + \texttt{Weight}[3] + \texttt{safter}[\texttt{last}[3]] - \texttt{sbefore}[7] = 0+3+15-17 = 1$,
- $\underline{\text{regret}}(3) = tlb_{\varnothing} + \texttt{Weight}[4] + tlb_{3,4,5,6,7} - tlb_{1,2,3,4,5,6,7}$
  $= \texttt{sbefore}[\texttt{first}[4]] + \texttt{Weight}[4] + \texttt{safter}[\texttt{last}[4]] - \texttt{sbefore}[7] = 0+10+15-17 = 8$,
- $\underline{\text{regret}}(4) = tlb_{\varnothing} + \texttt{Weight}[5] + tlb_{3,4,5,6,7} - tlb_{1,2,3,4,5,6,7}$
  $= \texttt{sbefore}[\texttt{first}[5]] + \texttt{Weight}[5] + \texttt{safter}[\texttt{last}[5]] - \texttt{sbefore}[7] = 0+6+15-17 = 4$,
- $\underline{\text{regret}}(5) = tlb_{\varnothing} + \texttt{Weight}[6] + tlb_{4,5,6,7} - tlb_{1,2,3,4,5,6,7}$
  $= \texttt{sbefore}[\texttt{first}[6]] + \texttt{Weight}[6] + \texttt{safter}[\texttt{last}[6]] - \texttt{sbefore}[7] = 0+6+11-17 = 0$,
- $\underline{\text{regret}}(6) = tlb_{\varnothing} + \texttt{Weight}[7] + tlb_{5,6,7} - tlb_{1,2,3,4,5,6,7}$
  $= \texttt{sbefore}[\texttt{first}[7]] + \texttt{Weight}[7] + \texttt{safter}[\texttt{last}[7]] - \texttt{sbefore}[7] = 0+9+10-17 = 2$,
- $\underline{\text{regret}}(7) = tlb_{1} + \texttt{Weight}[8] + tlb_{5,6,7} - tlb_{1,2,3,4,5,6,7}$
  $= \texttt{sbefore}[\texttt{first}[8]] + \texttt{Weight}[8] + \texttt{safter}[\texttt{last}[8]] - \texttt{sbefore}[7] = 3+5+10-17 = 1$,
- $\underline{\text{regret}}(8) = tlb_{1,2} + \texttt{Weight}[9] + tlb_{5,6,7} - tlb_{1,2,3,4,5,6,7}$
  $= \texttt{sbefore}[\texttt{first}[9]] + \texttt{Weight}[9] + \texttt{safter}[\texttt{last}[9]] - \texttt{sbefore}[7] = 3+10+10-17 = 6$.

Now that we know the lower regret of each value (for values 9 to 16 see column G of Fig. 3), we can use it for pruning the assignment variables according to the maximum value of the $Cost$ variable, which is equal to 18 in Example 3. We remove all values v for which the lower regret is strictly greater than 1 (e.g. the difference between the maximum cost 18 and the lower bound 17 we just compute in the previous section). Therefore we remove values 0, 1, 3, 4, 6, 8, 10, 12, 13 and 16 from variables $V_1, V_2, \ldots, V_e$.

Deriving a Better Lower Bound. We now show how to take advantage of the holes in the domains of the assignment variables and of the lower regret computed in this section to derive a sharper bound for the sum of the weights of the distinct values. The intuition behind this bound is as follows. For every assignment variable Var_i ($1 \leq i \leq n$) there exists at least one value v in its range for which the lower regret is equal to 0. However, if v does not belong to the domain of Var_i, then we may be forced to assign a value with a non-zero lower regret to Var_i, which would cause an increase of the lower bound. From the previous observation we get the following inequality $Cost \geq lower\_bound + \max\limits_{i \in 1,2,\ldots,n}\left(\min\limits_{v \in dom(Var_i)}\left(\underline{\text{regret}}(v)\right)\right)$, where $lower\_bound$ is the lower bound computed in Sect. 3.

5 A Tight Upper Bound for the Sum of the Weights of the Distinct Values

The purpose of this section is to show how to compute a tight upper bound for the sum of the weights of the distinct values. We give an incremental algorithm which can start from an arbitrary, possibly non-empty, matching between the assignment variables and the values. Throughout Sections 5 and 6, we illustrate the different phases of the corresponding algorithms on the instance given in the following example. Lines 1, 2 and 3 of Example 4 declare the set of potential values for each assignment variable as well as for the cost variable. Lines 4 to 11 state a *sum_of_weights_of_distinct_values* constraint where we have 16 assignment variables (see lines 4-5) that can take 21 potential values (see lines 6-11).

```
 1. V1,V2,V3::4, 8    V4::1, 4, 8,18    V5::1,11,18 V6:: 1, 5,11      V7::5,11
 2. V8      ::2, 5,10 V9::2,10          Va::2, 3,15 Vb:: 5, 6,7,13,19 Vc::6,7,13,19
 3. Vd      ::0,16,20 Ve::0, 9,16,17,19 Vf::9,14,17 Vg::12,20         Cost::138..200
 4. sum_of_weights_of_distinct_values({var-V1,var-V2,var-V3,var-V4,var-V5,
 5.  var-V6,var-V7,var-V8,var-V9,var-Va,var-Vb,var-Vc,var-Vd,var-Ve,var-Vf,var-Vg},
 6. {val-0  weight-13, val-1  weight-7 , val-2  weight-10, val-3  weight-3 ,
 7.  val-4  weight-10, val-5  weight-6 , val-6  weight-11, val-7  weight-11,
 8.  val-8  weight-15, val-9  weight-7 , val-10 weight-12, val-11 weight-4 ,
 9.  val-12 weight-5 , val-13 weight-14, val-14 weight-2 , val-15 weight-9 ,
10.  val-16 weight-3 , val-17 weight-8 , val-18 weight-5 , val-19 weight-5 ,
11.  val-20 weight-10},Cost)
```

Example 4. Instance used for illustrating Alg. 4, 5 and 6.

5.1 A Connection to Matching Theory

We first introduce the notion of *variable-value* graph G associated to an instance of the *sum_of_weights_of_distinct_values* constraint:

- For each assignment variable and each value that can be taken by at least one assignment variable we have exactly one vertex in G. So we can identify variables and values with the corresponding vertices in G. We will denote variable vertices by Var and value vertices by val.
- There is an edge $\{Var, val\}$ in G iff val is in the domain of Var.

A set M of edges is called a *matching* iff no two distinct edges $e, f \in M$ have a common vertex. Consider a vertex v. If there is an edge e in M that is incident to v, v is called *matched*, otherwise it is *free*. We assign a weight to every vertex of G as follows. Every variable vertex has weight zero, and every value vertex gets the weight that is assigned to the value in the constraint. The weight of M is defined as the sum of the weights of all matched vertices. Note that this differs from standard matching theory where one usually assigns weights to edges. M is called a *maximum weight matching* iff for all matchings M' in G we have $\text{weight}(M') \leq \text{weight}(M)$. Note that a maximum weight matching does not have to be of maximum cardinality.

Now we are ready to make the connection between the solutions of the constraint and maximum weight matchings in G. We assume that all variables $Var_1, Var_2, \ldots, Var_n$ are pairwise independent. This means, if we assign a value $val \in \text{dom}(Var_i)$ to Var_i, we can assign to Var_j $(j \neq i)$ any value in $\text{dom}(Var_j)$.

Proposition 4
The weight of a maximum weight matching in G is a tight upper bound on the maximum weight of a solution of the constraint, if all variables are pairwise independent.

Proof of Proposition 4
Let a denote an assignment of the variables and let $val_1, val_2, \ldots, val_k$ be the values used by a. For $i = 1,2,\ldots,k$ select a variable Var_i such that $a(Var_i) = val_i$. Then $\{\{Var_1, val_1\}, \{Var_2, val_2\}, \ldots, \{Var_k, val_k\}\}$ is a matching in G which has the same weight as the assignment. And hence, a maximum weight matching gives an upper bound on the weight of a solution.

What remains to show is that this bound is tight. Let M denote a maximum weight matching in G. Now we construct an assignment a. Let Var be a variable. If Var is matched with some value val, we set $a(Var) = val$. Otherwise Var is free and we set $a(Var)$ to some arbitrary value in $\text{dom}(Var)$. Since all values have non-negative weights, the weight of this assignment is at least $\text{weight}(M)$. As $\text{weight}(M)$ is an upper bound, we conclude that the weight of a equals the weight of M. □

We want to point out that this proposition does not hold anymore if negative weights are allowed or if the independence assumptions of the variables is violated. A maximum weight matching would still be an upper bound, but not necessarily a tight one.

So our task will be to develop an algorithm which computes a maximum weight matching in G efficiently. There are algorithms which can solve the problem by at most n shortest path computations. They can solve a more general problem where every edge of the graph has a weight, but the vertices have no weights. We want to take advantage of the special properties of our weights in order to come up with something more efficient. We will design an incremental algorithm with running time $O(n \cdot nedges)$ in the worst case, where $nedges$ is the number of edges of G. But in the best case the running time can go down to $O(nedges)$. Furthermore, we do not want to construct G explicitly, this point will be discussed later in detail after we have developed the algorithm.

5.2 Computing a Tight Upper Bound Incrementally

Some Results from Matching Theory. Before we describe our algorithm in the next section, we will summarize some important results from matching theory now. Let M denote an arbitrary matching in G. Consider a path p in G from a vertex x to a vertex y which uses the edges $e_1, e_2, \ldots, e_k$ in that order. We call p *alternating* with respect to M if the following holds:

– The edges of p are alternately in M and not in M.
– If $x = y$, then k is even (k may be 0).
– If $x \neq y$, then either $e_1 \in M$ or x is free, and either $e_k \in M$ or y is free.

If p is an alternating path with $x = y$, we call it an *alternating cycle*. In Fig. 4 we show examples of alternating paths and cycles.

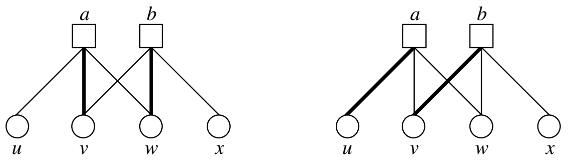


Fig. 4. Alternating paths: The edges of a matching M are shown in bold. On the left-hand side the paths p_1=[u,a,v,b,w] and p_2=[a,v,b,w] are alternating, c=[a,v,b,w,a] is an alternating cycle. But p_3=[u,a,v,b] is not alternating because it does not use the matching edge incident to b. The right-hand side shows the matching $M \oplus p_1$.

For two sets S_1 and S_2 we define the following exclusive-or operation $S_1 \oplus S_2 = (S_1 - S_2) \cup (S_2 - S_1)$. Then we can express two well-known results in matching theory:

– If p is an alternating path with respect to a matching M, then $M \oplus p$ is a matching again (cf. Fig. 4).
– If M and M' are both matchings in G, then $M \oplus M'$ is a collection of vertex-disjoint alternating paths and cycles.

Consider an alternating path p with respect to a matching M which connects two vertices x and y. We compare the weights of M and $M \oplus p$ by distinguishing four cases (see Fig. 5):

1. $x = y$:

   Then M and $M \oplus p$ match exactly the same vertices, and hence they have the same weight.

2. $x \neq y$ and both x and y are variable vertices:

   Then M and $M \oplus p$ match exactly the same value vertices, and hence they have the same weight.

3. $x \neq y$ and both x and y are value vertices:

Then one of the two vertices is free and the other one is matched in M. We assume that x is matched in M and y is free. Then $\text{weight}(M \oplus p) = \text{weight}(M) - \text{weight}(x) + \text{weight}(y)$.

4. $x \neq y$, one vertex is a variable vertex and one is a value vertex:

   We may assume that x is a variable vertex and y a value vertex. Either both vertices are free in M and $\text{weight}(M \oplus p) = \text{weight}(M) + \text{weight}(y)$, because $\text{weight}(x) = 0$. Or both vertices are matched and then $\text{weight}(M \oplus p) = \text{weight}(M) - \text{weight}(y)$.

We call p *augmenting* if $\text{weight}(M \oplus p) > \text{weight}(M)$. This can only be the case if one endpoint of p (let us say y) is a free value vertex and the other endpoint (let us say x) is a free variable vertex or a matched value vertex with $\text{weight}(x) < \text{weight}(y)$. When we state in an algorithm that we *augment* a matching M with an augmenting path p this means that we replace M by $M \oplus p$.

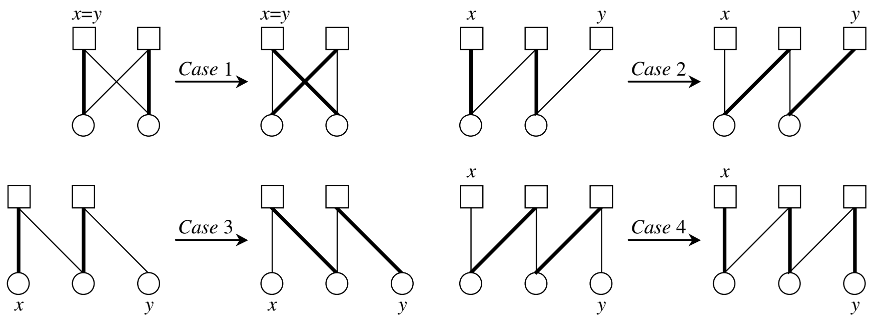


Fig. 5. The different cases for the relation of the matchings M and $M \oplus p$

Suppose we have two matchings M and M' with $\text{weight}(M) < \text{weight}(M')$. From the discussion above it is clear that $M \oplus M'$ must contain at least one augmenting path with respect to M.

Algorithm. Now we can develop our algorithm for computing a maximum weight matching in G (see Alg. 4). We start with an arbitrary matching M_{init}, which may be empty. The basic idea is to repeatedly look for augmenting paths and to augment the current matching M until it has maximum weight.

The search for an augmenting path works as follows. In the first phase we look for paths that start in a free variable vertex. We pick such a free vertex Var and grow a tree of alternating paths with Var as its root. We use breadth-first-search, which guarantees that every vertex in the tree is reached via a shortest alternating path. While we grow the tree we maintain a variable max_val which stores a free value vertex that has maximum weight among all free value vertices in the tree. If the tree contains no free vertex with positive weight, then $max\_val = none$. The search marks all vertices in the tree as *reached*.

When the tree is completed and we have found an augmenting path (i.e. $max\_val \neq none$), we augment the current matching M accordingly, and we abandon the tree by marking all vertices in it as *unreached* again. If the search terminates without finding an augmenting path, we know that all vertices in the tree cannot reach a free value vertex with positive weight. So when a later call to BFS hits such a vertex, it does not have to explore it again. And hence, we do not reset the marks of the vertices.

In the second phase of the search we process paths that start in a matched value vertex. It will turn out that start points for augmenting paths can only be those value vertices $val_1, val_2, \ldots, val_k$ which have already been matched in M_{init}. We observe that val_i is matched in M when it is considered, but it may have a different mate than in M_{init}. This is because during the first phase every matched vertex stays matched and during the second phase only the vertices that are considered before val_i can have changed their status from matched to free.

We process the vertices in increasing weight order, i.e. $\mathrm{weight}(val_1) \leq \mathrm{weight}(val_2) \leq \ldots \leq \mathrm{weight}(val_k)$. Thus the weights of the roots of the BFS-trees can only increase as the search goes on. This has the following consequence. When we grow a tree with root val_i without discovering an augmenting path, we know that all vertices in the tree can only reach free value vertices with weight at most $\mathrm{weight}(val_i)$. And hence, they do not have to be taken into account by later calls to BFS. So we can leave them marked *reached*.

We want to say a few words how we trace an augmenting path from max_val back to the root of its tree. While the BFS procedure grows a tree, it stores the father of every value vertex val in $father[val]$. The father of a variable vertex does not have to be stored because the father of a variable vertex is always the matching mate. With this information we can easily find the path from max_val to the root of its tree. We detect that we have reached the root when we either hit a free variable vertex (phase 1) or a value vertex val with $father[val] = none$ (phase 2).

We finish this section with a discussion of a possible optimization for Alg. 4. In some cases we can keep the tree although we have augmented the matching. Consider a call $\mathrm{BFS}(Var, w)$ and the tree T which is constructed by it. Suppose T contains before the augmentation only one value vertex val with weight greater than w. Then we do not have to destroy the tree, i.e. the marks of all value vertices in the tree can remain *reached*.

The optimization is valid because the following still holds:

Proposition 5
Consider a call $\mathrm{BFS}(Var, w)$ in phase one or two. Let M denote the matching after the call. If val is a value vertex with $mark[val] = reached$, then there is no alternating path wrt. M from val to a free value vertex val' with $\mathrm{weight}(val') > w$.

Proof of Proposition 5
First we prove by induction on the number of calls to BFS that the following invariant holds:

(I) If $\tilde{val}$ is a *reached* value vertex that is matched with $\tilde{Var}$, then all value vertices adjacent to $\tilde{Var}$ are marked *reached*.

The claim clearly holds before the first call. What remains to show is that it holds after a call if it holds before. So let M denote the matching after the call, and let T be the tree constructed during the call. If $\tilde{val}$ has been *reached* before, then $\tilde{val} \notin T$. And hence its mate $\tilde{Var}$ in M has been its mate before the call. Thus all adjacent value vertices have been *reached* before. So none of them belongs to T, and they are still marked *reached*.

In case $\tilde{val}$ has been marked by this call, we have that $\tilde{val}$ and its mate $\tilde{Var}$ belong to T and the marks of the vertices in T have not been reset. Thus it is clear that all neighbors of $\tilde{Var}$ are marked *reached*.

Now we can prove the statement of the proposition. Suppose the statement is false, i.e. there is a path p wrt. M from a *reached* value vertex val to a free value vertex val' with $\text{weight}(val') > w$. Since every suffix of p that starts in value vertex val is also an alternating path, we may assume that val is the only vertex on p which is marked *reached*. If p is not empty, it starts with the matching edge $\{val, Var\}$ and then uses a non-matching edge $\{Var, val''\}$. By the invariant (I), $mark[val''] = reached$, but this is a contradiction to the choice of p.

So p is empty, i.e. $val = val'$ and hence $mark[val'] = reached$. Let M' denote the matching before the call. We distinguish two cases depending on whether val' was matched in M'. Suppose first it was matched in M'. As it is free in M, this means that we are in the second phase and val' is the root of T. But then $\text{weight}(val') = w$, a contradiction. So val' must be free in M', which implies that it is marked *reached* by the last BFS call. We can also conclude that the optimization has been applied in this call, otherwise the mark of val' would have been reset, Thus val' is the only free vertex with weight greater than w in T, and hence we have $max\_val = val'$ for this BFS call. But then val' would be matched in M (after the augmentation), a contradiction. □

Correctness. In this section we prove the correctness of the algorithm. For this purpose, we will introduce some more terminology. Our algorithm starts with an initial matching M_{init}. Let $\mathcal{V}$ denote the set of value vertices that are matched in M_{init}. These vertices remain matched in the current matching M during phase 1. The matching M^1 that our algorithm has computed at the end of the first phase is in general not a maximum weight matching of G, otherwise we would not need the second phase. However, it will turn out that M^1 has maximum weight among all matchings of G where all the vertices in $\mathcal{V}$ are matched.

Thus we introduce the following notion. Let $\mathcal{V}$ denote a set of vertices and M a matching in a bipartite graph G. Then M is called a $\mathcal{V}$*-matching* iff all vertices in $\mathcal{V}$ are matched in M. Let p be an alternating path with respect to M from a vertex x to a vertex y. Then p is said to be $\mathcal{V}$*-alternating* iff $x = y$ or both x and y are not in $\mathcal{V}$. And if M is a weighted $\mathcal{V}$*-matching*, p is called $\mathcal{V}$*-augmenting* iff it is $\mathcal{V}$-alternating and augmenting with respect to M. Note that for $\mathcal{V} = \emptyset$ the terms

$\mathcal{V}$-matching and $\mathcal{V}$-alternating are identical to matching and alternating respectively. Some important properties carry over:

– If M is a $\mathcal{V}$-matching and p is a $\mathcal{V}$-alternating path, then $M \oplus p$ is a $\mathcal{V}$-matching.
– If M and M' are $\mathcal{V}$-matchings then $M \oplus M'$ is a collection of $\mathcal{V}$-alternating paths and cycles.
– Let M and M' be weighted $\mathcal{V}$-matchings with $\text{weight}(M) < \text{weight}(M')$. Then $M \oplus M'$ contains at least one $\mathcal{V}$-augmenting path wrt. M.

Our first milestone on the way to the correctness proof is the following proposition:

Proposition 6
Let M_{init} be an arbitrary matching in the variable-value graph G, and denote by $\mathcal{V}$ the set of matched value vertices. Then the first phase of Alg. 4 computes a maximum weight $\mathcal{V}$-matching in G.

Proof of Proposition 6
Let $Var_1, Var_2, \ldots, Var_k$ be the variable vertices in the order as they are considered by Alg. 4 in line 3 during the first phase. Let G_0 be the subgraph of G that is induced by all the value vertices and the variable vertices that are matched in $M_0 := M_{init}$. For $i = 1,2,\ldots,k$ we define G_i to be the subgraph of G induced by the vertices of G_{i-1} and the vertex Var_i. Denote by M_i the matching that is computed in the *i*-th iteration in phase 1. We will show by induction that M_i is a maximum weight $\mathcal{V}$-matching in G_i, which proves the claim because $G_k = G$.

We begin with the base case $i = 0$. Since the number of variable vertices in G_0 equals the cardinality of $\mathcal{V}$, M_0 is the only $\mathcal{V}$-matching in G_0. Note that M_0 is in general no maximum weight matching in G_0.

Now we come to the induction step. We assume that M_{i-1} is a maximum weight $\mathcal{V}$-matching of G_{i-1}. We want to show that M_i is a maximum weight $\mathcal{V}$-matching in G_i. First we prove that M_i is a $\mathcal{V}$-matching and $weight(M_i) \geq weight(M_{i-1})$. So assume that line 25 of Alg. 4 is executed in the *i*-th iteration, otherwise there is nothing to show. Clearly, the algorithm computes an alternating path p from Var_i to a free value vertex val with positive weight. Since $\mathcal{V}$ contains only value vertices and all of them are matched in M_{i-1}, we can conclude that p is $\mathcal{V}$-augmenting.

What remains to prove is that M_i has maximum weight in G_i among all $\mathcal{V}$-matchings. Let M' denote a maximum weight $\mathcal{V}$-matching in G_i. Then $M_{i-1} \oplus M'$ is a collection of vertex-disjoint $\mathcal{V}$-alternating paths and cycles. If none of these paths is augmenting, we are done. So let p denote a $\mathcal{V}$-augmenting path wrt. M_{i-1} in $M_{i-1} \oplus M'$. Since M_{i-1} is a maximum weight $\mathcal{V}$-matching in G_{i-1}, the path p cannot be a path in G_{i-1}, i.e. it visits Var_i. As Var_i is free in M_{i-1}, the vertex Var_i is an endpoint of p. Thus the other endpoint is a value vertex val with positive weight. As $M' \oplus p$ is a $\mathcal{V}$-matching in G_{i-1}, we have $\text{weight}(M' \oplus p) \le \text{weight}(M_{i-1})$ and we obtain:

$$\begin{aligned}\text{weight}(M') &= \text{weight}(M' \oplus p) + \text{weight}(val) \le \text{weight}(M_{i-1}) + \text{weight}(val)\\ &= \text{weight}(M_{i-1} \oplus p) \le \text{weight}(M') \ .\end{aligned}$$

Thus $\text{weight}(M_{i-1} \oplus p) = \text{weight}(M')$ and we may assume $M' = M_{i-1} \oplus p$. Since the algorithm explores in its *i*-th iteration all alternating paths that start in Var_i, it cannot miss p. And hence, M_i is a maximum weight $\mathcal{V}$-matching in G_i. □

Based upon Proposition 6 we can prove the correctness of our algorithm:

Proposition 7
Algorithm 4 computes a maximum weight matching in G.

Proof of Proposition 7
Let $val_1, val_2, \ldots, val_k$ be the value vertices in the order as they are considered by the algorithm in line 6 of Alg. 4 during the second phase. Let M_0 be the matching at the beginning of the second phase and denote by M_i the matching computed in the *i*-th iteration. For $i = 0,1,\ldots,k$ define $\mathcal{V}_i$ to be the set $\{val_{i+1}, val_{i+2}, \ldots, val_k\}$. We intend to prove by induction that M_i is a maximum weight $\mathcal{V}_i$-matching in G for all i. This implies the claim because $\mathcal{V}_k$ is empty.

The case $i = 0$ follows immediately from the previous proposition, for $\mathcal{V}_0$ contains exactly those value vertices that are matched in M_{init}.

Now we come to the induction step. Let us assume that M_{i-1} is a maximum weight $\mathcal{V}_{i-1}$-matching in G. We have to show that M_i is a maximum weight $\mathcal{V}_i$-matching in G. First we convince ourselves of the fact that M_i is a $\mathcal{V}_i$-matching with $\text{weight}(M_i) \ge \text{weight}(M_{i-1})$. Since $\mathcal{V}_i \subset \mathcal{V}_{i-1}$, we have that M_{i-1} is also a $\mathcal{V}_i$-matching. If the algorithm augments M_{i-1} in the *i*-th iteration it does so by an alternating path from val_i to a free value vertex val such that $\text{weight}(val_i) < \text{weight}(val)$. Observe that both val_i and val are not in $\mathcal{V}_i$. Thus the path is $\mathcal{V}_i$-augmenting.

Now we show that M_i has maximum weight among all $\mathcal{V}_i$-matchings. Similar as in the previous proof, we consider a maximum weight $\mathcal{V}_i$-matching M'. We see that if M_{i-1} is not a maximum weight $\mathcal{V}_i$-matching, then there must be a $\mathcal{V}_i$-augmenting path p in $M_{i-1} \oplus M'$. Furthermore val_i must be an endpoint of p, otherwise p would also be $\mathcal{V}_{i-1}$-augmenting. Since val_i is matched in M_{i-1}, the other endpoint of p is a free value vertex val with $\text{weight}(val) > \text{weight}(val_i)$. A similar argument as in the previous proof shows that we can assume $M' = M_{i-1} \oplus p$. Since the algorithm cannot miss p, we conclude that M_i is a maximum weight $\mathcal{V}_i$-matching in G. □

Running Time and Implementation. Now let us analyze the running time. Let n denote the number of variable vertices and $nedges$ the number of edges of G. Thus $nedges$ is the sum of the sizes of all variable domains. We assume that we have list L_{init} that contains all the values matched in M_{init} sorted in increasing weight order. It is clear that this list can be constructed in time $O(card(M_{init}) \cdot \log(card(M_{init})))$. Later we will discuss how L_{init} can be maintained in an incremental setting.

Since no vertex marks are reset between two consecutive augmentations, we can conclude that the time between two augmentations is $O(nedges)$, no matter how many unsuccessful attempts we make in between. The time for an augmentation is clearly also $O(nedges)$. So the total running time is $O((a+1) \cdot nedges)$, where a is the number of augmentations. We show that this number is bounded by the cardinality c of the final matching. Let a_1, a_2 denote the number of augmentations in the first and the second phase of the algorithm respectively. We see that every augmentation in the first phase increases the cardinality of the matching by one and in the second phase the cardinality remains constant. So we have $c = a_1 + \mathrm{card}(M_{init}) \geq a_1 + a_2 = a$, for the number of augmentations in the second phase is bounded the cardinality of the initial matching. Therefore $O(n \cdot nedges)$ is an upper bound for the running time.

We can make some interesting observations. The matching M^1, which the algorithm has computed at the end of phase one, does not have to be of maximum cardinality. This is because we ignore values of weight zero. Let G^+ denote the graph which is obtained by removing all value vertices of weight zero from G. So all value vertices in G^+ have strictly positive weight. Thus any cardinality-augmenting path in G^+ is also weight-augmenting. And hence M^1 and the final matching have maximum cardinality in G^+. This implies that c is independent of M_{init}.

If we start with an empty matching, then the algorithm skips phase two and it makes c augmentations. Even if we run the algorithm with a "bad" initial matching, we cannot have more augmentations. But if we start with a "good" matching, i.e. one which matches nearly all the value vertices of the final matching, we can be much faster. And if the initial matching already has maximum weight, the algorithm runs in time $O(nedges)$ because there can be no augmentations. Therefore, we believe that the algorithm is suitable for settings where incrementality is important.

Now we discuss how to maintain the list L_{init} which contains the matched values of M_{init} sorted in increasing weight order. What we will actually show is how the algorithm can be extended such that it outputs not only the maximum weight matching M, but also a list L containing the matched values in M in sorted order. The running time of the extended algorithm will still be $O((a+1) \cdot nedges)$. The algorithm constructs L as follows. It starts with am empty list and whenever an augmentation occurs (see line 25 of Alg. 4), it appends *max_val* to L_{new}. In addition, it removes[11] from L_{init} every value that changes its state from matched to free during an augmentation in the second phase. Thus at any time, $L_{init} \cup L_{new}$ contains exactly the matched values of M. In order to construct L when phase two has terminated, we sort L_{new} and then we merge the two sorted lists L_{init} and L_{new}.

[11] L_{init} and L_{new} are doubly linked lists, and for every matched value we store the address of the corresponding list-item, so we can delete an item in $O(1)$.

Let us analyze the time for this computation. The maintenance of L_{init} and L_{new} requires only constant overhead per augmentation. When phase two terminates, the cardinality of L_{new} is equal to a, and $nedges$ is an upper bound on the cardinality of L_{init}. Sorting L_{new} requires $O(a \log a)$ time and merging takes time $O(a + nedges)$. Observing that $a \le nedges$, we get the claimed time bound for the extended algorithm.

Let us discuss some details concerning the integration of the algorithm in the constraint programming framework. As we have already pointed out, we do not have to construct the graph G explicitly, for the only operation that we need is to scan all value vertices that are adjacent to a variable vertex. This amounts to iterating through all values in the domain of a given variable. In many constraint programming systems the time required for this operation is linear in the size of the domain[12]. In this case the complexity analysis for the graph algorithm carries over to the constraint programming setting. However, we have to discuss some detail in the implementation. We must associate some information with every value, namely its *mark* and its *father*. Since the values are (possibly) large integers, we cannot use a simple array for this purpose. But we can use perfect hashing [8] to construct a data structure which allows to access the information associated with a given value in worst-case time $O(1)$. The worst-case space requirement is linear in the number of values, and the average time to build the structure is also linear.

When the algorithm is called for the first time, we run it on an empty matching[13] M_{init}. (Thus L_{init} is also empty.) We store the output matching M and the ordered list L of matched values. During the search process of the constraint programming system, some infeasible values may be removed from the variable domains. This means that some edges disappear from G. So before we run the algorithm again, we remove from M all edges that correspond to erased values, and we update L accordingly. Thus we obtain a new − usually non-empty − initial matching M'_{init} and a list L'_{init} for the algorithm. In order to update these data structures we have to consider every edge $\{Var, val\}$ in M and query the constraint programming system if val is still contained in $\mathrm{dom}(Var)$. We want to point out that this may not be a constant time operation.

We also update M and L upon backtracking since we don't trail modifications on M and L in order to save memory.

Let us make a final observation. We have seen that the matching computed by the algorithm does not have to yield a tight upper bound, if there is some dependence among the variables. Clearly, we cannot exclude all dependencies because the constraint is in general used together with other constraints on its variables. But we can recognize if the user of the constraint mentions the same variable more than once in the argument list of the constraint or if some variables are identified during the

[12] This is for instance true in SICStus Prolog where a domain is implemented by a list of intervals of consecutives values. But this is not true when one uses an array of bits.

[13] One could use a greedy algorithm for computing M_{init}.

search. For this purpose we sort the variables according to their ids[14] and remove duplicates before we run our algorithm.

```
PROCEDURE COMPUTE_MAX_WEIGHT_MATCHING(M_init)
set M to M_init;
initialize mark of all value vertices as unreached;

PHASE 1:
FOR ALL free variable vertices Var wrt. M_init DO
    tree:=∅;
    BFS(Var,0);

PHASE 2:
FOR ALL value vertices val matched in M_init in increasing weight order DO
    IF mark[val]=unreached THEN
        mark[val]:=reached; tree:={val}; father[val]:=none;
        let Var denote the mate of val wrt. M;
        BFS(Var,weight(val));
RETURN M;

PROCEDURE BFS(Var,w)
max_val:=none;
set queue Q to [Var];
WHILE Q not empty DO
    extract first vertex Var from Q;
    FOR ALL value vertices val' adjacent to Var DO
        IF mark[val']=unreached THEN
            mark[val']:=reached; tree:=tree∪{val'}; father[val']:=Var;
            IF val' is matched wrt. M THEN
                let Var' denote the matching mate of val';
                append Var' to Q;
            ELSE IF weight(val')>w THEN
                max_val:=val'; w:=weight(max_val);
IF max_val≠none THEN
    augment M with path from root to max_val (with the aid of father);
    FOR ALL value vertices val in tree DO
        mark[val]:=unreached;
```

Algorithm 4: Computation of a maximum weight matching in *G*

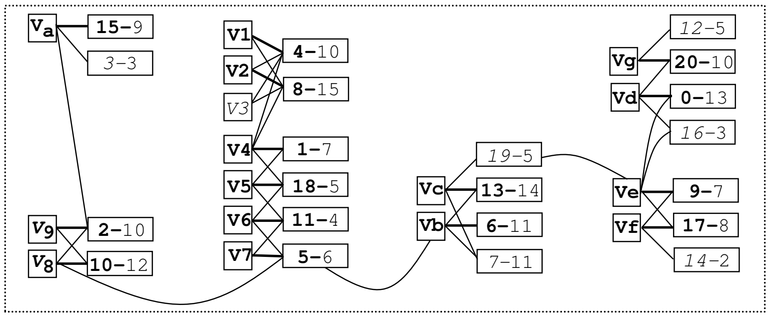


Fig. 6. Matching of maximum weight associated to Example 4

Fig. 6 shows the "variable-value" graph *G* associated to the *sum_of_weights_of_distinct_values* constraint given in Example 4. The vertices associated to the variables are displayed with a

[14] Often a variable is represented as a pointer to the actual data structure. So the address of the target would be a suitable id for the variable.

square and the label of the corresponding variable, while the vertices connected to values are shown with a rectangle which contains the value followed by its weight. The matched vertices of the maximum matching leading to the upper bound of 141 are shown in bold, while those edges that belong to the maximum matching are marked with a thick line. Using Alg. 4 on Example 4 produces the following maximum matching $V_2 = 8(15)$, $V_c = 13(14)$, $V_d = 0(13)$, $V_8 = 10(12)$, $V_b = 6(11)$, $V_9 = 2(10)$, $V_1 = 4(10)$, $V_g = 20(10)$, $V_a = 15(9)$, $V_f = 17(8)$, $V_e = 9(7)$, $V_4 = 1(7)$, $V_7 = 5(6)$, $V_5 = 18(5)$, $V_6 = 11(4)$, where weights are shown in parenthesis after each value. The maximum matching has an overall cost of 15+14+13+12+11+10+10+10+9+8+7+7+6+5+4=141 which gives a tight upper bound for the $Cost$ variable of Example 4.

6 Computing the Upper Regret

This section first introduces the notion of *upper regret*, denoted $\overline{\text{regret}}(Var, val)$, associated to a pair Var, val where Var is an assignment variable and val a value in $\text{dom}(Var)$. Then it provides an algorithm in order to prune the assignment variables according to this upper regret and to the minimum possible value of the $Cost$ variable. The *upper regret* $\overline{\text{regret}}(Var, val)$ for the pair $\{Var, val\}$ is the minimum decrease of the upper bound for the sum of the weights of the distinct values under the hypothesis that variable Var is assigned to value val.

Translated to our matching scenario the upper regret can be interpreted like this. Let M denote a maximum weight matching in G, and let M' denote a maximum weight matching under the restriction $e = \{Var, val\} \in M'$. We have $\text{weight}(M') \leq \text{weight}(M)$ and $\overline{\text{regret}}(Var, val) = \text{weight}(M) - \text{weight}(M') \geq 0$.

We make some observations which will help us to compute the upper regret for e. If $e \in M$ then $\overline{\text{regret}}(Var, val) = 0$. So let us assume $e \notin M$. Thus $M \oplus M'$ contains e. Let $p = p(e)$ denote the alternating path in $M \oplus M'$ that uses e and has maximum length. Let x and y be the start and the end vertex of p respectively. We want to show that $\text{weight}(M \oplus p) = \text{weight}(M')$, i.e. $M \oplus p$ has also maximum weight among all matchings that contain e. Let us define $\tilde{w}_x$ and $\tilde{w}_y$ as follows. If p starts with an edge in M', set $\tilde{w}_x = \text{weight}(x)$, otherwise $\tilde{w}_x = -\text{weight}(x)$. We define $\tilde{w}_y$ in an analogous way depending on whether p ends with an edge in M'. Then we get

$$\begin{aligned} \text{weight}(M') &= \text{weight}(M' \oplus p) + \tilde{w}_x + \tilde{w}_y \\ &\leq \text{weight}(M) + \tilde{w}_x + \tilde{w}_y = \text{weight}(M \oplus p) \leq \text{weight}(M') . \end{aligned}$$

If $x = y$ then p is an alternating cycle, which implies $\text{weight}(M \oplus p) = \text{weight}(M)$, and hence the upper regret is zero. Now we examine the case $x \neq y$. We may assume that p visits Var before val. So we can decompose p as $p = p_x \circ e \circ p_y$ such that p_x is a path from x to Var and p_y is a path from val to y (see Fig. 7). We observe that p_x is empty or ends with an edge in M, and p_y is either empty or starts with an edge in M. Thus p_x, p_y are alternating paths with respect to M. If p_x is empty or starts with an edge in M', then x is a free variable vertex wrt. M, otherwise x is a matched value vertex. In any case, we have $\text{weight}(M \oplus p_x) = \text{weight}(M) - \text{weight}(x)$. If p_y is empty or ends with an edge in M', then y is a free value vertex wrt. M, otherwise y is a matched variable vertex. In either case, we obtain $\text{weight}(M \oplus p_y) = \text{weight}(M) - \text{weight}(val) + \text{weight}(y)$. Putting the results together we get $\text{weight}(M \oplus p) = \text{weight}(M) - \text{weight}(x) + \text{weight}(y)$. And thus $\overline{\text{regret}}(Var, val) = \text{weight}(x) - \text{weight}(y)$.

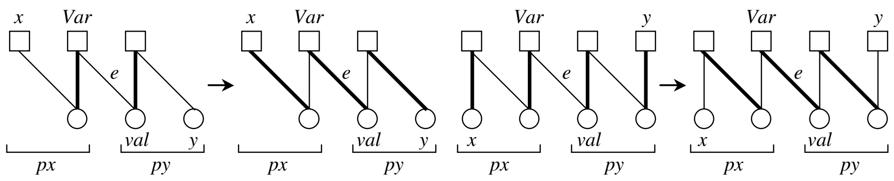


Fig. 7. The different possibilities for the path *p*(*e*): *x* is either a free variable or a matched value, and *y* is either a free value or a matched variable.

Now we are ready to develop our algorithm. We will not consider the matching M' anymore, the terms "matched" and "free" will always refer to M. It will turn out helpful to construct a directed graph $\vec{G}$ from the undirected graph $G = (V, E)$ and the matching M. The graph $\vec{G} = (V, \vec{E})$ has the same vertices as G and the following edges (cf. Fig. 8):

- For every edge $\{Var, val\}$ in E we have the directed edge (Var, val) in $\vec{E}$.
- For every edge $\{Var, val\}$ in M we have the directed edge (val, Var) in $\vec{E}$.

This means, for every undirected edge we have a directed edge from the variable to the value vertex, and for every matching edge we have in addition an edge directed in the opposite direction.

This graph is interesting because alternating paths in G with respect to M correspond to directed paths in $\vec{G}$:

- Alternating cycles in G correspond to simple cycles in $\vec{G}$ of length greater than two, and vice versa. Simple cycles of length two can be identified with matching edges.
- There is a one-to-one correspondence between acyclic alternating paths in G and simple paths in $\vec{G}$ which start in a free variable or matched value vertex and end in a matched variable or free value vertex.

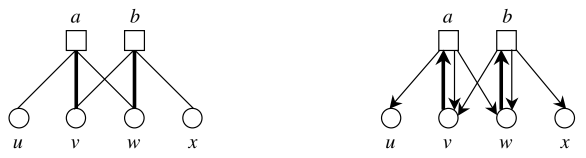


Fig. 8. The right-hand side shows the directed graph $\vec{G}$ for the graph *G* and the matching *M* on the left-hand side. Observe how the alternating path *p*=[*w*,*b*,*v*,*a*,*u*] in *G* translates to a directed path in $\vec{G}$.

In order to make the description of the algorithm easier, let us introduce an abbreviation. We say that a vertex x *can reach* a vertex y iff there is a, possibly empty, path from x to y in $\vec{G}$, in formulas we write $x \to^* y$ to denote this.

The basic ideas for the algorithm are as follows. If we want to determine the upper regret for an edge $e = \{Var, val\}$, we first check whether Var and val lie on a cycle in $\vec{G}$. If this is the case the upper regret is 0, and we are done. Otherwise, $e \notin M$ and there exists an alternating path $p(e)$ in G which passes through e such that its start vertex x and end vertex y determine the upper regret (see the discussion above). Since we assumed that $p(e)$ visits Var before val, it translates to a directed path $\vec{p}(e) = \vec{p}_x \circ (Var, val) \circ \vec{p}_y$ in $\vec{G}$ from x to y (see again Fig. 7). As we have already seen, the upper regret of e is the difference $\mathrm{weight}(x) - \mathrm{weight}(y)$.

We can observe that the weight of the start vertex of $\vec{p}(e)$ only depends on the variable vertex Var of e, but not on val. Let us denote this weight by $label(Var)$. We see that $label(Var)$ is zero, if Var can be reached from a free variable vertex. Otherwise it is the minimum weight of all value vertices that can reach Var.

On the other hand, the weight of the end vertex of $\vec{p}(e)$ is completely determined by the value vertex val of e. We use $label(val)$ to denote this weight. It is the maximum weight of a free value vertex that val can reach. If val can reach only matched value vertices, then the end vertex of $\vec{p}(e)$ is a matched variable, and hence $label(val)$ is zero.

Since we want to compute the upper regret for every edge, we label every vertex v in $\vec{G}$ with $label(v)$. And then the upper regret of an edge $e = \{Var, val\}$ is simply the difference $label(Var) - label(val)$. Thus the algorithm consists of four steps:

1. Compute the strongly connected components of $\vec{G}$:
   We assign component numbers to every vertex such that two vertices receive the same number iff they belong to the same strongly connected component.

2. Label the variable vertices:
   We label every variable vertex Var with $label(Var)$:
   $$label(Var) = \begin{cases} 0, \text{ if there is a free variable } Var' \text{ with } Var' \to^* Var \\ \min\{\mathrm{weight}(\mathrm{val}) : val \to^* Var\}, \text{ otherwise} \end{cases}$$

3. Label the value vertices:
   We determine for every value vertex val its label $label(val)$:
   $$label(val) = \max(\{0\} \cup \{weight(val') : val \to^* val' \wedge val' \text{ is a free value}\})$$

4. Compute the upper regret:
   For every edge $e = \{Var, v\}$ we compute the upper regret as follows:
   $$\overline{\mathrm{regret}}(Var, val) = \begin{cases} 0, & \text{if } Var \text{ and } val \text{ belong to same SCC} \\ label(Var) - label(val), & \text{otherwise} \end{cases}$$

We prove the correctness of the algorithm above. Fix an edge $e = \{Var, val\}$. If $e \in M$, then the upper regret is zero and Var, val belong to the same strongly connected component, because the edges (Var, val) and (val, Var) are both in $\vec{G}$. Otherwise we observe that the algorithm will not miss the path $p(e)$, thus it computes a value which is not larger than the real upper regret.

What remains to show is that we can find a matching M' which contains e such that $\text{weight}(M) - \text{weight}(M')$ is the value computed by the algorithm for the upper regret. If Var and val belong to the same strongly connected component, then there is a simple cycle $\vec{c}$ containing the two vertices. If the length of $\vec{c}$ is two, then $e \in M$ and we choose $M' = M$. Otherwise $\vec{c}$ translates to an alternating cycle c. Thus $M' = M \oplus c$ contains e and has the same weight as M.

If the two vertices lie in different strongly connected components, we can find a simple path $\vec{p}_x$ from some vertex x to Var which made us label Var with $label(Var)$. (Thus x is a free variable vertex or a matched value vertex.) And there is a simple path $\vec{p}_y$ from val to some vertex y which made the algorithm assign the label to val. (So y is a free value vertex or − if val can only reach matched variable vertices − some arbitrarily chosen variable vertex that can be reached from val.) The path $\vec{p} = \vec{p}_x \circ (Var, val) \circ \vec{p}_y$ in $\vec{G}$ must be simple because Var and val are in different strongly connected components. Thus it translates to an alternating path p in G. And for $M' = M \oplus p$ we have $\text{weight}(M') = \text{weight}(M) - \text{weight}(x) + \text{weight}(y)$, and hence $\text{weight}(M) - \text{weight}(M') = label(Var) - label(val)$. □

In the sequel we will show how to implement all the four steps of the algorithm in time $O(nedges)$[15]. For the final step this is obvious. For the first step we notice that there are algorithms for computing the strongly connected components of a directed graph in linear time. Some of these algorithms only have to scan the outgoing edges of the vertices of the graph, but not the incoming edges (see for example [5]).

So the critical steps are the labelling steps. We give the pseudo-code for these steps in Alg. 5 and 6.

In order to determine the labels of the variable vertices we do not consider those vertices one by one and compute the label for each of it, for this would take too long. But we do it the other way round. We take the first free variable vertex Var and start a depth-first-search which assigns the label 0 to all variable vertices that Var can reach. Then we consider the next free variable vertex, start a depth-first-search from there and label all reachable vertices that have not been labelled yet, and so on. After that we pick a value vertex val with smallest weight, and we use another depth-first-search to label all reachable vertices that are still unlabelled with $\text{weight}(val)$. We repeat this procedure until we have considered all value vertices in increasing order of their weights.

Note that whenever the search encounters a labelled vertex, it does not have to explore it because its label and the labels of all its descendants are not greater than the current label. This guarantees that every edge is scanned only once.

[15] We assume again that we already have a sorting of all values according to their weights.

```
PROCEDURE COMPUTE_VAR_LABELS
initialize the labels of all variable vertices to unreached;
FOR ALL free variable vertices Var of $\vec{G}$ DO VAR_DFS(Var,0);
FOR ALL matched value vertices val of $\vec{G}$ in increasing weight order DO
    let Var denote the mate of val; VAR_DFS(Var,weight(val));
PROCEDURE VAR_DFS(Var,w)
IF label(Var)=unreached THEN
    label(Var):=w;
    FOR ALL value vertices val′ adjacent to Var DO
        IF val′ has a matching mate Var′ THEN VAR_DFS(Var′,w);
```

Algorithm 5: Computation of the variables vertex labels

Now we discuss the labelling of the value vertices as shown in Alg. 6. One could use the same idea as in the previous algorithm: Pick a free value vertex val with maximum weight and label all value vertices which can be reached from val with $\mathrm{weight}(val)$. Repeat this step until all free value vertices have been processed in decreasing weight order. The problem of this approach lies in the term “can be reached from val”, i.e. the approach requires to scan the incoming edges of the vertices of $\vec{G}$. So for a value val we would ask for every variable Var with $val \in \mathrm{dom}(Var)$. This is a query which is not supported efficiently by constraint programming systems. So we decided to take another approach that requires only to scan the outgoing edges of a vertex.

Our algorithm is based on depth-first-search. We will recall some properties of depth-first-search which are important for us. Every vertex in the graph has an exploration status which is one of the following three values: *unreached*, *active* and *completed*. A vertex v has status *unreached* until the procedure depth-first-search is called for it. Then its status changes from *unreached* to *active*, and when the call for v terminates its status becomes *completed* and remains this way till the end of the algorithm. A top-level call to a vertex v explores all vertices that are reachable from v and that are marked *unreached* at the time of the call. After the call all vertices that are reachable from v are *completed* which means that depth-first-search has visited all of them.

This gives rise to the first version of our depth-first-search algorithm which differs slightly from the final version in Alg. 6, but it suffices to sketch the main ideas. We will later discuss its short-coming and how to remedy them.

We assume that at the beginning of the algorithm all value vertices have the label zero. In a call $\mathrm{DFS}(val)$ for an *unreached* value vertex we do the following. After changing the status of val to *active*, we check whether val is free. If this is the case, then we label it with $\mathrm{weight}(val)$ and terminate the call, because val has no outgoing edge. Otherwise, there is only one outgoing edge (val, Var), which corresponds to the matching edge $\{Var, val\} \in M$. We do not make a recursive call for Var, but we scan every outgoing edge (Var, val') in the call for val. If val' is *unreached*, we make a recursive call $\mathrm{DFS}(val')$. In any case we set the label of val to the maximum of its current label and the label of val'. When all edges out of Var are scanned, the call for val is finished. Upon termination we mark val as *completed*.

When a top-level call $\mathrm{DFS}(val)$ terminates, val is labelled correctly with $label(val)$. But the value vertices visited by recursive calls may receive a label smaller than $label(val)$. This can be seen in Fig. 9. We have two variables Var_1, Var_2 and three values val_1, val_2, val_3. The vertex val_3 is free and has weight 1. Assume we make the call $\mathrm{DFS}(val_1)$. We mark val_1 as *active* and scan the edges out of its mate Var_1. This leads to a recursive call $\mathrm{DFS}(val_2)$ which makes val_2 active and scans the edge (Var_2, val_1). Since val_1 is *active*, we do nothing and leave the label of val_2 at zero. Then the call for val_2 terminates after marking it as *completed*. So we return to the top-level and call $\mathrm{DFS}(val_3)$. As val_3 is free, we set the label of val_3 to 1 and return. Then we adjust the label of val_1 to 1. And the top-level call also terminates. So val_1 is labelled correctly. But the label of val_2 is zero, although this vertex can also reach val_3. The problem is that val_1 does not have its final label when the edge (Var_2, val_1) is scanned. The problem only occurs when the status of the target of the scanned edge is *active*. There is no harm if the status is *unreached* at the beginning of the scan, because then depth-first-search is invoked on the target, so that it is explored and marked as *completed* before the adjustment on the label.

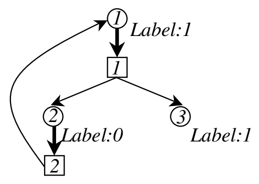


Fig. 9. Example for the fact that the first DFS version does not produce the correct labels for all vertices. (Values are drawn as circles, variables as boxes.)

The following observations will help us to solve the problem. At any time the vertices that are currently *active* lie on a single path that ends in the currently explored vertex val. So if we scan an edge into an active vertex val', then val and val' lie on a directed cycle, i.e. they belong to the same strongly connected component. It is clear that val and val' as well as all other vertices in their strongly connected component can reach exactly the same vertices in $\vec{G}$. Let val'' be the value vertex in the strongly connected component of val that is visited first by depth-first-search. Consider now the point in time when val'' becomes completed. Then val'' cannot reach any *active* vertex, because such a vertex would belong to the same strongly connected component as val'' and it would have been visited before val''. And val'' cannot reach any vertex with mark *unreached*, this is a property of depth-first-search. Thus val'' is labelled correctly. We will later give a rigorous proof of this fact, but let us now finish the development of our algorithm. We use basically the same algorithm as above, but we do not assign a label to every value vertex anymore, instead of this we assign a value label to every strongly connected component of $\vec{G}$ such that $val\_label[scc(val)] = label(val)$ for every value vertex val. Thus we obtain Alg. 6.

```
 1 PROCEDURE COMPUTE_VAL_LABELS
 2 compute strongly connected components of $\vec{G}$ for setting scc of each value;
 3 initialize the labels of all strongly connected components to 0;
 4 initialize mark of all value vertices to unreached;
 5 FOR ALL value vertices val of $\vec{G}$ DO
 6 |  IF mark[val]=unreached THEN VAL_DFS(val);

 7 PROCEDURE VAL_DFS(val)
 8 mark[val]:=active;
 9 IF val is free THEN val_label[scc(val)]:=weight(val);
10 ELSE
11 |  let Var denote the matching mate of val;
12 |  FOR ALL vertices val' adjacent to Var DO
13 |  |  IF mark[val']=unreached THEN VAL_DFS(val');
14 |  |  val_label[scc(val)]:=max(val_label[scc(val)],val_label[scc(val')]);
15 mark[val]:=completed;
```

Algorithm 6: Computation of the value vertex labels

Now we prove the correctness of Alg. 6. It is easy to see that the labels can never become too big, i.e. after the initialization in line 3 we have $0 \le val\_label[scc(val)] \le label(val)$ for every value vertex val. Let us say that the label of val is *tight* if $val\_label[scc(val)] = label(val)$. Since labels can only increase, we see that once a label becomes tight, it stays tight till the end of the algorithm.

The algorithm ignores free variable vertices because they have only outgoing edges and each of them forms a strongly connected component of its own. So let Var denote a matched variable vertex and let val be its mate. Let us call an edge $\vec{e} = (Var, val')$ in $\vec{G}$ *alive* until it is scanned in the for-loop and line 14 is executed for val and val', afterwards $\vec{e}$ is called *dead*. In order to make the following statements easier, we also call an edge *alive* if it is directed from a value to a variable vertex (i.e. if it corresponds to a matching edge). Consider a strongly connected component of $\vec{G}$. The first vertex in it that is made *active* by VAL_DFS is called the *root of the component*[16].

Now we can formulate the main fact which immediately implies the correctness of the algorithm:

Proposition 8

Let r denote the root of a strongly connected component of $\vec{G}$. While r is marked active, at least one of the following holds[17]*:*

1. *There is a (possibly empty) path $\vec{p}$ from r to a free value vertex val with $weight(val) = label(r)$ such that all edges on $\vec{p}$ are alive.*
2. *There is a (possibly empty) path $\vec{q}$ from r to a value vertex val' with $val\_label[scc(val')] = label(r)$ such that all edges on $\vec{q}$ are alive.*

When r is marked completed, its label is tight.

[16] This name was introduced in [5]; it reflects the fact that all other vertices of this component become descendants of this vertex in the DFS tree.

[17] We want to point out that none of the statements has to hold as long as *r* is *unreached*, it may be the case that all paths of alive edges end in vertices that have not received their final label then.

Before we prove the statement, let us discuss the intuition behind it. Let us assume that $label(r) > 0$, otherwise there is nothing to show. Then there exists a path $\vec{p}$ from r to some free value vertex val with weight $label(r)$ by the definition of $label(r)$. Consider a point in time when r is *active*: Maybe we have not explored any edge on the path yet, i.e. all edges on $\vec{p}$ are still alive (cf. case 1). But it can also be that we have already explored a suffix of $\vec{p}$ that starts in a value vertex val'. Then the label of val' is tight and the prefix $\vec{q}$ of $\vec{p}$ from r to val' uses only alive edges (see case 2). Note that $\vec{q}$ maybe empty, which means that the label of r is tight.

Proof of Proposition 8

First we show that the statement about *active* roots implies that the label of a *completed* root r is tight. If r is free, then it has no outgoing edges in $\vec{G}$ and its label is tight because line 9 (the statement after `THEN`) has been executed for it. In the case that r is matched, we observe that all outgoing edges of r are dead. When its last edge died, r was still *active*. Since any path from r to a free value vertex is non-empty, we conclude that the label of r must have been tight at that moment.

Now we come to the statement about *active* roots and we will prove it by induction. We assume w.l.o.g. $label(r) > 0$ and show that the claim holds at the moment when r is made *active* and continues to hold – as long as r is *active* – whenever an edge in $\vec{G}$ becomes dead. So consider the point in time when r changes its mark from *unreached* to *active*. Let $\vec{p}$ be a path from r to a free value vertex val with $\text{weight}(val) = label(r)$. If all edges on $\vec{p}$ are alive we are done. So assume that $\vec{e} = (Var', val'')$ is the first dead edge on $\vec{p}$, and let val' denote the mate of Var'. We will show that the label of val' is tight.

Denote by r'' the root of the component of val''. We go back to the point in time when $\vec{e}$ dies in the call $VAL\_DFS(val')$ and distinguish three cases depending on the mark of r'' at that time:

- $mark[r''] = unreached$:

  Since val'' is not *unreached* at that time, the root r'' must have already been reached, too.
- $mark[r''] = completed$:

  So the label of r'' (and hence of val'') is tight, i.e. $val\_label[scc(val'')] = label(r)$. Thus when $\vec{e}$ dies as line 14 is executed, $val\_label[scc(val')]$ is set to $label(r)$.
- $mark[r''] = active$:

  This implies that r'' lies on the path of *active* vertices, which ends in val'. So val', val'' and r'' are in the same strongly connected component. If r'' is *completed* before r is made *active*, we know the labels of r'' and hence val' are tight when r gets marked *active*. Assume now that r'' is still *active* when r becomes *active*. Then r'' can reach r. Since val'' is on $\vec{p}$ and r'' is in the strongly connected component of val'', we can infer that r can reach r''. So r and r'' are roots of the same strongly connected component, and hence $r = r''$. This is a contradiction to the fact that r is *unreached*, but r'' is *active* when $\vec{e}$ dies.

We come to the induction step. The interesting case is the following. We have a path $\vec{p}$ from r to some value vertex val, $\vec{p}$ consists only of alive edges, and $val\_label[scc(val)] = label(r)$ or val is a free vertex with weight $label(r)$. Now some edge $\vec{e} = (Var', val'')$ on $\vec{p}$ dies because line 14 is executed during VAL_DFS(val'), where val' is the matching mate of Var'.

As before we consider the root r'' of the strongly connected component of val''. The cases that r'' is *unreached* or *completed* can be handled with the same arguments as above. Only the case that r'' is marked *active* when $\vec{e}$ dies must be treated in a different way. Consider the path $\vec{a}$ of *active* vertices that ends in val'. It visits both r and r''. Suppose $\vec{a}$ contains a sub-path from r'' to r. Since r can reach val'' and hence r'', all the three vertices belong to the same strongly connected component, which implies $r = r''$. So $\vec{a}$ surely contains a (possibly empty) sub-path from r to r''. Since r'' is the root of its component, $\vec{a}$ cannot visit val'' before r''. Therefore there is a (possibly empty) sub-path $\vec{a}'$ from r to Var''. Now we observe that $\vec{a}$ uses only alive edges. Let $\vec{p}_{tail}$ denote the suffix of $\vec{p}$ from Var'' to val. Then $\vec{a} \circ \vec{p}_{tail}$ is a path from r to val which consists only of alive edges. □

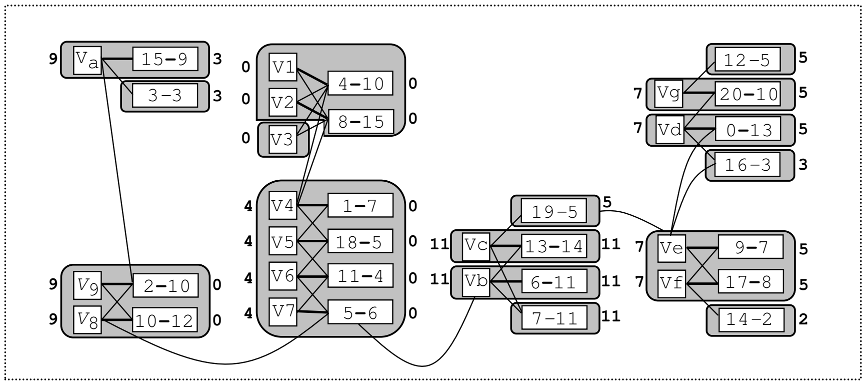


Fig. 10. Strongly connected components of $\vec{G}$ and label of the variables and values vertices

Fig. 10 shows the graph $\vec{G}$ associated to the maximum matching given in Fig. 6 and its strongly connected components. The edges from a variable vertex to a value vertex are shown with a standard line, while a thick line displays those edges that go in both directions. In addition we show for each variable and value vertex its label in bold.

Here is the detail of the computation of the upper regret for those edges that do not belong to the matching of maximum weight given in Fig. 6.

- $\overline{\text{regret}}(V_3,8) = 0$, $\overline{\text{regret}}(V_3,4) = 0$.
- $\overline{\text{regret}}(V_4,4) = 4-0 = 4$, $\overline{\text{regret}}(V_4,8) = 4-0 = 4$.
- $\overline{\text{regret}}(V_8,5) = 9-0 = 9$.
- $\overline{\text{regret}}(V_a,2) = 9-0 = 9$, $\overline{\text{regret}}(V_a,3) = 9-3 = 6$.
- $\overline{\text{regret}}(V_b,5) = 11-0 = 11$, $\overline{\text{regret}}(V_b,7) = 11-11 = 0$, $\overline{\text{regret}}(V_b,13) = 11-11 = 0$.

- $\overline{\text{regret}}(V_c,19) = 11-5 = 6$, $\overline{\text{regret}}(V_c,7) = 11-11 = 0$.
- $\overline{\text{regret}}(V_d,16) = 7-3 = 4$, $\overline{\text{regret}}(V_d,20) = 7-5 = 2$.
- $\overline{\text{regret}}(V_e,0) = 7-5 = 2$, $\overline{\text{regret}}(V_e,16) = 7-3 = 4$.
- $\overline{\text{regret}}(V_f,14) = 7-2 = 5$.
- $\overline{\text{regret}}(V_g,12) = 7-5 = 2$.

Finally, we remove a value v from a variable Var if the difference between the upper bound computed in Sect. 5 and the upper regret $\overline{\text{regret}}(Var,v)$ is strictly less than $\min(Cost)$.

Since, in Example 4, the minimum value of the $Cost$ variable is 138 and since the upper bound computed for the $Cost$ variable was equal to 141, we remove all edges $\{V,val\}$ for which the quantity $141-\overline{\text{regret}}(V,val)$ is strictly less than 138. This leads to remove 4 and 8 from V_4, 5 from V_8, 2 and 3 from V_a, 5 from V_b, 19 from V_c, 16 from V_d, V_e and 14 from V_f.

7 Linking both Sides of the Constraint

This section presents three deduction rules. The first rule is useful when the domain of the $Cost$ variable contains holes, namely when we require the cost to take some specific target value within a given set of possible values. It combines the lower and upper regrets, which were respectively introduced in Sect. 4 and 6. The second rule is valuable when both sides of the $Cost$ variable are constrained and partially answer a question arising in the conclusion of the article about the *global cardinality constraint with costs* [13]. The third rule is helpful when we want to improve the lower bound of the $Cost$ variable according to some additional constraints between the assignment variables. All these three rules are in fact generic and can be applied for any cost-filtering algorithm where we can compute the lower and upper regret associated to the fact that we fix a specific assignment variable to a given value[18]. To illustrate their wide applicability, we first recall other existing constraints for which we can effectively apply these rules. We then present the three rules.

Beside the *sum_of_weights_of_distinct_values* constraint introduced in this article the deduction rules of this section can be applied to the following constraints:

– The *minimum weight alldifferent* constraint (see [3], [7], [14]).
– The *minimum weight alldifferent* constraint with a restriction on the maximum number of cycles [4][19].
– The *global cardinality constraint with costs* [13].

As for the *sum_of_weights_of_distinct_values* constraint, we generalize these constraints so as to enforce the $Cost$ variable to be equal to the cost associated to the assignment variables. We now present the first generic rule, which allows further

[18] Within this section, n stands for the number of assignment variables and m for the number of distinct values that can be taken by these variables.

[19] In [4] the maximum number of cycles is restricted to one.

pruning of the assignment variables according to both the lower and upper regret, as well as to the holes of the $Cost$ variable.

Proposition 9
Let $lower\_bound$ and $\underline{\text{regret}}(Var,v)$ be respectively a lower bound computed for the $Cost$ variable and the corresponding lower regret according the fact that the assignment variable Var is fixed to value v. Similarly, let $upper\_bound$ and $\overline{\text{regret}}(Var,v)$ be an upper bound for $Cost$ and the corresponding upper regret.

If $[lower\_bound + \underline{\text{regret}}(Var,v), upper\_bound - \overline{\text{regret}}(Var,v)] \cap \text{dom}(Cost) = \varnothing$

then $v \notin \text{dom}(Var)$.

Proof of Proposition 9
From the definitions of the lower and upper regrets, the complement of interval $[lower\_bound + \underline{\text{regret}}(Var,v), upper\_bound - \overline{\text{regret}}(Var,v)]$ is the set of impossible values for the $Cost$ variable under the assumption that the assignment variable Var is fixed to v. Therefore if the interval does not intersect the domain of the $Cost$ variable, value v should be removed from variable Var. □

We come to the second generic rule, which allows considering simultaneously, both the smallest and largest values of the $Cost$ variable.

Proposition 10
Let $lower\_bound$ and $\underline{\text{regret}}(Var,v)$ be respectively a lower bound computed for the $Cost$ variable and the corresponding lower regret according the fact that the assignment variable Var is fixed to value v. For a given non-negative integer r, denote by $upper\_bound_r$ the upper bound computed for the $Cost$ variable under the hypothesis that the domain of each assignment variable Var is restricted to the set of values $\{v : \underline{\text{regret}}(Var,v) \leq r\}$. Denote by $rmin$ the smallest value such that $[lower\_bound, upper\_bound_{rmin}] \cap \text{dom}(Cost) \neq \varnothing$. Then an improved lower bound for the $Cost$ variable is $lower\_bound + rmin$.

In order to locate $rmin$, we perform a binary search in a table of distinct lower regrets. This leads to a worst-case complexity of $O(n \cdot m \cdot \log(n \cdot m) + \log r \cdot U)$, where $O(U)$ stands for the worst-case complexity for evaluating an upper bound of the cost and r to the number of distinct lower regrets. The contribution $O(n \cdot m \cdot \log(n \cdot m))$ corresponds to the sort of the lower regrets for creating a table of distinct lower regrets. Finally we mention that a proposition similar to Proposition 10 allows to further update the upper bound of the $Cost$ variable.

As an illustration of direct application of this rule, consider the question raised in [13] of finding a general algorithm in order to partially take into account both the minimum and the maximum value of the cost variable in the case of the *global cardinality constraint with costs*. In the conclusion of his article, Régin illustrates this question with the following instance of an infeasible problem: "consider the problem involving three variables x_1, x_2 and x_3 with $D(x_1) = \{a,b\}$, $D(x_2) = \{b,c\}$ and $D(x_3) = \{a,c\}$. Each value has to be taken at most 1. A cost function is defined as follows: $cost(x_1,a) = cost(x_2,b) = cost(x_3,c) = 1$ and $cost(x_1,b) =$

$cost(x_2,c)=cost(x_3,a)=3$. The sum of any instantiation of all the variables must be greater than 4 and less than 8.". In a first step we compute the lower bound and the lower regret of this problem using the filtering algorithm described in [13]. This gives a lower bound of 3 (denoted $lower\_bound$) as well as the following lower regret: $\underline{\text{regret}}(x_1,a)=\underline{\text{regret}}(x_2,b)=\underline{\text{regret}}(x_3,c)=0$ and $\underline{\text{regret}}(x_1,b)=\underline{\text{regret}}(x_2,c)=\underline{\text{regret}}(x_3,a)=6$. Since $upper\_bound_0$ is equal to 0, we have that $[lower\_bound, upper\_bound_0] \cap \text{dom}(Cost)=\varnothing$. For the next possible smallest lower regret 6, we have $upper\_bound_6=9$. Now, since $[lower\_bound, upper\_bound_6] \cap \text{dom}(Cost)\neq\varnothing$, Proposition 10 tells us that $lower\_bound+6=9$ is a lower bound for the cost. Since the maximum value of the cost is 8 we have a contradiction.

Finally we present the third generic rule, which allows estimating a better lower bound according to additional constraints relating the assignment variables. One typical application of this rule is the *minimum weight all different* constraint [14], where in addition we want to take partially into account the fact that we have a constraint on the maximum number of cycles of the corresponding permutation.

Proposition 11
Let $lower\_bound$ and $\underline{\text{regret}}(Var,v)$ be respectively a lower bound computed for the $Cost$ variable and the corresponding lower regret according the fact that the assignment variable Var is fixed to value v. Furthermore assume that the assignment variables $Var_1,Var_2,\ldots,Var_n$ have also to satisfy a conjunction of constraints $\mathcal{C}$ for which we use an algorithm $A(\mathcal{C},Var_1,Var_2,\ldots,Var_n)$ in order to check whether $\mathcal{C}$ is infeasible or whether $\mathcal{C}$ may be feasible according to the domains of the assignment variables. Denote by $rmin$ the smallest value such that, when restricting the domain of each assignment variable Var_i $(1\le i\le n)$ to the values $\{v:\underline{\text{regret}}(Var_i,v)\le rmin\}$, $A(\mathcal{C},Var_1,Var_2,\ldots,Var_n)$ does not detect the infeasibility of $\mathcal{C}$. Then an improved lower bound for the $Cost$ variable is $lower\_bound+rmin$.

The search of $rmin$ is performed in the same way as in Proposition 10, where we perform a binary search. Finally we mention that a proposition similar to Proposition 11 allows to further update the upper bound of the $Cost$ variable.

As direct application of Proposition 11, consider the *minimum weight all different* constraint [14], where the assignment variables have to be pairwise distinct, and where the $Cost$ variable is equal to the sum of the costs associated to the arcs (Var_i,v_i) (v_i is the value taken by the *i*-th assignment variable). Furthermore, assume that we have an additional constraint on the maximum number of cycles $maxc$ of the permutation $\langle Var_1,Var_2,\ldots,Var_n\rangle$. In this context, the algorithm $A(\mathcal{C},Var_1,Var_2,\ldots,Var_n)$ of Proposition 11 can be implemented as follows. Consider the directed graph G, with n vertices and an arc between the *i*-th and the *j*-th vertices when both $j\in\text{dom}(Var_i)$ and $\underline{\text{regret}}(Var_i,j)\le rmin$. Algorithm A detect the infeasibility of the constraint on the maximum number of cycles by computing the number of strongly connected components of G and by checking that it is less than or equal to $maxc$. So now we get a lower bound of the *Cost* variable, which partially[20] takes into account the fact that we don't want to have more than *maxc* cycles.

8 Positioning of the *Sum of Weights of Distinct Values* Constraint

This section first positions the *sum_of_weights_of_distinct_values* constraint among existing constraints as well as according to some combinatorial problems. Then it mentions a heuristic derived from the fact that Alg. 1, 2, 3, 4, 5 and 6 achieve a complete pruning when all variables of the *sum_of_weights_of_distinct_values* constraint are interval variables.

Generalization of Existing Constraints. The *sum_of_weights_of_distinct_values* constraint generalizes the following constraints:

- The $alldifferent(\{V_1,V_2,\ldots,V_n\})$[21] constraint [10] is obtained by setting all the weights to one and by fixing the cost variable to the number of assignment variables n.
- A relaxation of the $alldifferent(\{V_1,V_2,\ldots,V_n\})$ constraint where one counts the minimum number C of variables for which the value needs to be changed in order that all variables take a distinct value [11]. This constraint is obtained by setting all the weights to one and the cost variable to $n-C$.
- Finally, the number of distinct values constraint $nvalue(N,\{V_1,V_2,\ldots,V_n\})$[22] [2] is obtained by setting all the weights to one and the cost variable to N.

From a pruning point of view, the *sum_of_weights_of_distinct_values* filters out all values removed by the complete filtering algorithms of the *alldifferent* [12] and of the relaxed *alldifferent* [11] constraints. It also subsumes the pruning provided for the *nvalue* constraint in [2] without changing the worst-case complexity of the original algorithm.

Modeling Domination Problems. We now show how to use the *sum_of_weights_of_distinct_values* constraint in order to model domination problems. Given a graph G with vertices V and edges E, the problem is to find a subset S of V such that for every vertex $u \in V-S$ there exists a vertex $v \in S$ such that u is adjacent to v. Usually one wants to minimize the number of elements of S. In their book [9], T. W. Haynes, et al. provide a good reference on domination problems. Capturing the structure of domination problems with one single global constraint is important from at least two points of view:

[20] Partially, since the described algorithm checks only a necessary condition for having no more than *maxc* cycles.

[21] The $alldifferent(\{V_1,V_2,\ldots,V_n\})$ constraint holds if all variables $V_1,V_2,\ldots,V_n$ are pairwise different.

[22] The $nvalue(N,\{V_1,V_2,\ldots,V_n\})$ constraint holds if N is the number of distinct values taken by variables $V_1,V_2,\ldots,V_n$.

– From a practical point of view, domination problems occur in a lot of real-world applications[23] like assignment problems as well as computer communication networks.
– From a theoretical point of view one could hope to enhance the computation of the lower bound as well as the pruning according to the maximum value of the cost variable. One way would be to try to reuse lower bounds or algorithms from domination theory. Within constraint programming this was hardly possible since a domination problem was broken into a large number of elementary constraints.

We now come to the model. First we give to each vertex of G a unique identifier, which is an integer. We then associate to each vertex $v \in V$ of G a domain variable D_v for which the domain consists of the identifier of v as well as of the identifiers of those vertices that are adjacent to v. When D_v is fixed, its value is interpreted as the vertex that dominates v. All the variables D_v are put in the first argument of the *sum_of_weights_of_distinct_values* constraint, while their potential values are placed in the second argument with a weight of one. Finally, the domain of the *Cost* variable (i.e. the third argument of the constraint) is set up according to the required size of the set S.

Modeling the Cost of Some Assignment Problems. In several assignment problems, one usual component of the cost can be directly expressed with the *sum_of_weights_of_distinct_values* constraint without introducing any extra variable. This is for instance the case for the warehouse location problem [16] where one part of the cost corresponds to the sum of the individual costs of the warehouses that effectively supply at least one customer. This cost is modeled as follows. To each warehouse we associate a natural number ranging from 1 to the total number of warehouses m. For each customer we create a domain variable $Customer_i$ and initialize its domain to the warehouses, which can effectively supply him. Finally we set up the following constraint

$$\text{sum\_of\_weights\_of\_distinct\_values}\left(\begin{array}{l}\{var-Customer_1, var-Customer_2, \ldots, var-Customer_n\}, \\ \left\{\begin{array}{l} val-1 \;\; weight-CostWarehouse_1, \\ val-2 \;\; weight-CostWarehouse_2, \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ val-m \;\; weight-CostWarehouse_m \end{array}\right\}, \; CostWarehouses \end{array}\right),$$

where $CostWarehouse_i$ $(1 \le i \le m)$ is the cost for opening the *i*-th warehouse. In the third argument of the constraint, the domain variable *CostWarehouses* expresses directly the total cost related to the warehouses that are really used.

Heuristics for the *sum_of_weights_of_distinct_values* Constraint. The fact that Alg. 1, 2, 3, 4, 5 and 6 perform a complete pruning when the domains of all variables are intervals suggests the following heuristics for trying out the different values of an assignment variable. Instead of trying out each possible value of an assignment variable, we only need to restrict that variable to each of its intervals of consecutive

[23] Chapter 1 of [9] provides a good overview of concrete applications of domination problems.

values. When we have done this for all assignment variables we know that the *sum_of_weights_of_distinct_values* can for sure be satisfied.

9 Conclusion

In this article, we have introduced a new constraint which allows expressing directly the fact that a cost variable is the sum of the weights associated to the distinct values taken by a given set of assignment variables. For this constraint we came up with two classes of cost-filtering algorithms with low complexities:

- The first class estimates a lower bound for the cost variable and prunes the assignment variables according to this lower bound as well as to the maximum allowed cost. Since we could not come up with one single complete algorithm, we develop several algorithms for different special cases. In particular, we came up with a complete filtering algorithm when all the domains of the assignment variables are intervals.
- The second class estimates an upper bound for the cost variable and prunes the assignment variables according to this upper bound as well as to the minimum allowed cost. For this side of the constraint, we came up with one complete filtering algorithm.

Finally, we proposed three generic deduction rules relating both aspects of this family of cost constraints. One important result is the ability to mechanically enhance the bounds of the cost variable by taking into account additional constraints between the assignment variables.

The different algorithms described in this article were implemented within SICStus Prolog in order to provide the *sum_of_weights_of_distinct_values* constraint and to enhance the *global cardinality constraint with costs* and the *minimum weight alldifferent* constraint with a restriction on the maximum number of cycles. As far as we know, it is the first time that all aspects of the cost variable are taken into account for pruning the other variables. This allows to provide a truly declarative constraint which reacts to every kind of modifications on the cost variable and which can therefore be applied to a wider range of problems.

Acknowledgements

Thank to Per Mildner for useful comments on an early version of this article.

References

1. Baptiste, P., Le Pape, C., Peridy, L.: Global Constraints for Partial CSPs: A Case-Study of Resource and Due Date Constraints. In *Principles and Practice of Constraint Programming - CP'98*, 4th International Conference, Pisa, Italy, (October 26–30, 1998), Proceedings. Lecture Notes in Computer Science, Vol. 1520, Springer, 87–101, (1998).

2. Beldiceanu, N.: Pruning for the minimum Constraint Family and for the number of distinct values Constraint Family. In *Principles and Practice of Constraint Programming - CP'2001*, 7th International Conference, Paphos, Cyprus, (November 26 – December 1, 2001). Proceedings. Lecture Notes in Computer Science, Vol. 2239, Springer, 211−224, (2001).
3. Caseau, Y., Laburthe, F.: Solving Various Weighted Matching Problems with Constraints. In *Principles and Practice of Constraint Programming - CP'97*, 3rd International Conference, Schloss Hagenberg, Austria, (October 29 − November 1, 1997), Proceedings. Lecture Notes in Computer Science, Vol. 1330, Springer, 17−31, (1997).
4. Caseau, Y., Laburthe, F.: Solving Small TSPs with Constraints. In *Fourteenth International Conference on Logic Programming - ICLP'97*, Leuven, Belgium, (July 8−11, 1997), Proceedings. MIT Press, 316−330, (1997).
5. Cheriyan, J., Mehlhorn, K.: Algorithms for Dense Graphs and Networks on the Random Access Computer. *Algorithmica*, 15(5), 521−549, (1996).
6. Fahle, T.: Cost Based Filtering vs. Upper Bounds for Maximum Clique. In *Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'02)*, 93−107, Le Croisic, France (March 25−27, 2002).
7. Focacci, F., Lodi, A., Milano, M.: Cost-Based Domain Filtering. In *Principles and Practice of Constraint Programming - CP'99*, 5th International Conference, Alexandria, Virginia, USA, (October 11−14, 1999), Proceedings. Lecture Notes in Computer Science, Vol. 1713, Springer, 189−203, (1999).
8. Fredman, M.,L., Komlós, J., Szemerédi, E.: Storing a Sparse Table with $O(1)$ Worst Case Access Time. *Journal of the ACM*, 31 (3), 538−544, (1984).
9. Haynes, T.W., Hedetniemi, S. T., Slater, P. J.: *Fundamentals of domination in graphs*. Marcel Dekker, Inc., (1998).
10. Laurière, J.-L.: A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence* 10, 29−127, (1978).
11. Petit, T., Régin, J.-C., Bessière, C.: Specific Filtering Algorithms for Over-Constrained Problems. In *Principles and Practice of Constraint Programming - CP'2001,* 7th International Conference, Paphos, Cyprus, (November 26 − December 1, 2001). Lecture Notes in Computer Science, Vol. 2239, Springer, 451−463, (2001).
12. Régin, J.-C.: A filtering algorithm for constraints of difference in CSP. In *Proc. of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, 362−367, (1994).
13. Régin, J.-C.: Arc Consistency for Global Cardinality Constraints with Costs. In *Principles and Practice of Constraint Programming - CP'99*, 5th International Conference, Alexandria, Virginia, USA, (October 11−14, 1999), Proceedings. Lecture Notes in Computer Science, Vol. 1713, Springer, 390−404, (1999).
14. Sellmann, M.: An Arc Consistency Algorithm for the Minimum Weight All Different Constraint. In *Principles and Practice of Constraint Programming - CP'2002*, 8th International Conference, Cornell University, Ithaca, NY, USA, (September 8−13, 2002), Proceedings. Lecture Notes in Computer Science, Vol. 2470, Springer, 744−749, (2002).
15. Van Hentenryck, P., Carillon J.-P.: Generality Versus Specificity: An Experience with AI and OR Techniques. In *Proceedings of the American Association for Artificial Intelligence (AAAI-88)*, (St. Paul, MN), AAAI, Menlo Park, Calif., (August, 1988).
16. Van Hentenryck, P.: *The OPL Optimization Programming Language*. The MIT Press, Cambridge, MA, (1999).