Pruning for the *minimum* Constraint Family and for the *Number of Distinct Values* Constraint Family

Nicolas Beldiceanu

SICS

Lägerhyddsvägen 18
SE-75237 Uppsala, Sweden
Email: nicolas@sics.se

November 10 2000
SICS Technical Report T2000/10
ISSN 1100-3154
ISRN: SICS-T--2000/10-SE

Abstract The paper presents propagation rules that are common to the *minimum* constraint family and to the *number of distinct values* constraint family. One original contribution is to provide a geometrical interpretation of these rules that can be used by a generic sweep pruning algorithm. Finally one practical interest of the paper is to describe an implementation of the *number of distinct values* constraint. This is a quite common counting constraint that one encounters in many practical applications such as timetabling or frequency allocation problems.

Keywords Global constraint, Minimum, Number of distinct values.

1 Introduction

The purpose of this paper is to present propagation rules for the *minimum* constraint family that was introduced in [1]. The *minimum* constraint family has the form $\text{minimum}(M, r, \{V_1, \dots, V_n\})$ where M is a variable, r is an integer value ranging from 0 to $n-1$, and $\{V_1, \dots, V_n\}$ is a collection of variables. Variables take their value in a finite discrete set of items. The constraint holds if M corresponds to the item of rank r according to a given total ordering relation $\Re$ between the items assigned to variables $V_1, \dots, V_n$ ¹. If there is no such item of rank r , M takes the maximum possible value over all items. Relation $\Re$ is defined in a procedural way by the following functions that will be used in order to make our propagation algorithms generic:

- min_item returns an item that corresponds to a value that is less or equal than all items that can be taken by variables $V_1, \dots, V_n$,
- max_item returns an item that corresponds to a value that is greater or equal than all items that can be taken by variables $V_1, \dots, V_n$,
- $I \prec J$ is true iff item I is less than item J ,
- $I \succ J$ is true iff item I is greater than item J ,
- $\text{next}(I)$: if $I \neq \text{max_item}$ then returns the smallest item that is greater than item I ,
- $\text{prev}(I)$: if $I \neq \text{min_item}$ then returns the largest item that is smaller than item I ,
- $\text{min}(V)$ returns the minimum item that can be assigned to variable V ,
- $\text{max}(V)$ returns the maximum item that can be assigned to variable V ,
- $\text{remove_val}(V, I)$ removes item I from the feasible values of variable V ,
- $\text{adjust_min}(V, I)$ adjusts the minimum feasible value of variable V to item I ,
- $\text{adjust_max}(V, I)$ adjusts the maximum feasible value of variable V to item I .

Defining a member C of the *minimum* constraint family will be achieved by providing the previous set of functions for the total ordering relation $\Re$ that is specific to constraint C . This has the main advantage that one can introduce a new member of the family without having to reconsider all the propagation algorithms. The complexity results about the algorithms of this paper assume that all functions used for defining $\Re$ are performed in $\mathcal{O}(1)$.

The next section presents some instances of the *minimum* constraint family. Sections 3 and 4 present two algorithms that are used several times by the different pruning algorithms. These algorithms provide a lower bound for the minimum number of distinct values and for the $(r+1)^{\text{th}}$ smallest distinct value. Section 5 shows how to reduce the domain of variable M , while Section 6 explains how to shrink domains of variables $V_1, \dots, V_n$. Section 7 shows how to reinterpret the deduction rules of the previous section in order to define the *minimum* constraint family in terms of

¹ This is different from the problem of finding the $(r+1)^{\text{th}}$ smallest value [3, pages 185-191]: in our case all the variables that have the same value have the same rank and we want to find the $(r+1)^{\text{th}}$ smallest distinct value. For instance, the second smallest distinct value of 9,4,1,3,1,4 is equal to 3 (and not 1).

forbidden regions [2]. Finally, the last section indicates how to use the algorithms of this paper in order to implement the propagation for the *number of distinct values* constraint.

2 The *minimum* Constraint Family

This section lists some instances of the *minimum* constraint family and provides the corresponding functions, which define the total ordering relation $\Re$, for two of the specified instances. Examples of the *minimum* family are:

- $\text{minimum}(MIN, \{VAR_1, \dots, VAR_n\})$: MIN is the minimum value of $VAR_1, \dots, VAR_n$,
- $\text{maximum}(MAX, \{VAR_1, \dots, VAR_n\})$: MAX is the maximum value of $VAR_1, \dots, VAR_n$,
- $\text{min_n}(MIN, r, \{VAR_1, \dots, VAR_n\})$: MIN is the minimum of rank r of $VAR_1, \dots, VAR_n$, or max_item if there is no variable of rank r ²,
- $\text{max_n}(MAX, r, \{VAR_1, \dots, VAR_n\})$: MAX is the maximum of rank r of $VAR_1, \dots, VAR_n$, or min_item if there is no variable of rank r ,
- $\text{minimum_pair}(PAIR, \{PAIR_1, \dots, PAIR_n\})$: $PAIR$ is the minimum pair of $PAIR_1, \dots, PAIR_n$,
- $\text{maximum_pair}(PAIR, \{PAIR_1, \dots, PAIR_n\})$: $PAIR$ is the maximum pair of $PAIR_1, \dots, PAIR_n$.

In all the previous constraints, MIN , MAX and $VAR_1, \dots, VAR_n$ are domain variables³, while $PAIR$ and $PAIR_1, \dots, PAIR_n$ are ordered pairs of domain variables. The next table gives for the *maximum* and *minimum_pair* constraints the different functions introduced in the first section. For *minimum_pair* $.x$ and $.y$ indicate respectively the first and second attribute of a pair, while MIN_Y and MAX_Y are the minimum and maximum value for the $.y$ attribute. MININT and MAXINT correspond respectively to the minimum and maximum possible integers. $\text{min_var}(V)$ (respectively $\text{max_var}(V)$) returns the minimum (respectively maximum) value of the domain variable V . $\text{remove_val_var}(V, I)$ removes value I from variable V . $\text{adjust_min_var}(V, I)$ (respectively $\text{adjust_max_var}(V, I)$) adjusts the minimum (respectively maximum) value of variable V to value I .

² Note that, removing value max_item from the possible values of variable MIN , will enforce the minimum of rank r to be defined.

³ A domain variable is a variable that ranges over a finite set of integers; $\text{min}(V)$ and $\text{max}(V)$ respectively denote the minimum and maximum values of variable V .

Table 1. Functions associated to the *maximum* and *minimum_pair* constraints

function \ constraint	<i>maximum</i>	<i>minimum_pair</i>
min_item	MAXINT	(MININT,MININT)
max_item	MININT	(MAXINT,MAXINT)
$I < J$	$I > J$	$(I.x < J.x) \vee (I.x = J.x \wedge I.y < J.y)$
$I > J$	$I < J$	$(I.x > J.x) \vee (I.x = J.x \wedge I.y > J.y)$
next(I)	$I - 1$	IF $I.y < \text{MAX_Y}$ THEN $(I.x, I.y + 1)$ ELSE $(I.x + 1, \text{MIN_Y})$
prev(I)	$I + 1$	IF $I.y > \text{MIN_Y}$ THEN $(I.x, I.y - 1)$ ELSE $(I.x - 1, \text{MAX_Y})$
min(V)	max_var(V)	(min_var($V.x$), min_var($V.y$))
max(V)	min_var(V)	(max_var($V.x$), max_var($V.y$))
remove_val(V, I)	remove_val_var(V, I)	IF $V.x = I.x$ THEN ⁴ remove_val_var($V.y, I.y$) IF $V.y = I.y$ THEN ⁵ remove_val_var($V.x, I.x$)
adjust_min(V, I)	adjust_max_var(V, I)	adjust_min_var($V.x, I.x$) IF max_var($V.x$) = $I.x$ THEN ⁶ adjust_min_var($V.y, I.y$)
adjust_max(V, I)	adjust_min_var(V, I)	adjust_max_var($V.x, I.x$) IF min_var($V.x$) = $I.x$ THEN ⁷ adjust_max_var($V.y, I.y$)

3 Computing a Lower Bound of the Minimum Number of Distinct Values of a Sorted List of Variables

This section describes an algorithm that evaluates a lower bound of the minimum number of distinct values of a set of variables $\{U_1, \dots, U_n\}$ sorted on increasing minimum value. Note that this is similar to the problem of finding a lower bound on

⁴ For the if conditional statement we should generate the constraint: $V.x = I.x \Rightarrow V.y \neq I.y$.

⁵ For the if conditional statement we should generate the constraint: $V.y = I.y \Rightarrow V.x \neq I.x$.

⁶ For the if conditional statement we should generate the constraint: $V.x = I.x \Rightarrow V.y \geq I.y$.

⁷ For the if conditional statement we should generate the constraint: $V.x = I.x \Rightarrow V.y \leq I.y$.

the number of vertices of the dominating set [6, page 190], [5, page 232] of the graph $G=(V,E)$ defined in the following way:

- to each variable of $\{U_1, \dots, U_n\}$ and to each possible value that can be taken by at least one variable of $\{U_1, \dots, U_n\}$ we associate a vertex of the set V ,
- if a value v can be taken by a variable U_i ($1 \leq i \leq n$) we create an edge that starts from v and ends at U_i ; we also create an edge between each pair of values.

We now give the algorithm:

```

1  ndistinct:=1;
2  reinit:=1;
3  i:=1;
4  WHILE i<n DO
5    i:=i+1-reinit;
6    IF reinit OR low < min( $U_i$ ) THEN low:=min( $U_i$ ); ENDIF;
7    IF reinit OR up > max( $U_i$ ) THEN up:=max( $U_i$ ); ENDIF;
8    reinit:=(low>up);
9    ndistinct:=ndistinct+reinit;
10 ENDWHILE;
```

Fig. 1. Generated intervals

Figure 1 shows the execution of the previous algorithm on a set of 9 variables $\{U_1, \dots, U_9\}$ with the respective domain 0..3, 0..1, 1..7, 1..6, 1..2, 3..4, 3..3, 4..6 and 4..5. Each variable 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, while intervals $low..up$ that are produced by the algorithm (see lines 6,7) are dashed. In this example the computed minimum number of distinct values is equal to 3.

The algorithm partitions the set of variables $\{U_1, \dots, U_n\}$ in $ndistinct$ groups of consecutive variables by starting a new group each time $reinit$ is set to value 1 (see line 8). If for each group we consider the variable with the smallest maximum value and the largest minimum value in case of tie, then we have $ndistinct$ pairwise non-intersecting⁸ variables. From this fact we derive that we have a valid lower bound. In the example of Figure 1 we have the three following groups

⁸ Two domain variables are called *non-intersecting variables* when they don't have any value in common.

U_1, U_2, U_3, U_4, U_5 and U_6, U_7 and U_8, U_9 . The three pairwise non-intersecting variables are variables U_2 , U_7 and U_9 . The lower bound obtained by the algorithm is sharp when for each group of variables there is at least one value in common. This is for example the case when each domain variable consists of one single interval of consecutive values. Note that the same algorithm works also if the set of variables $\{U_1, \dots, U_n\}$ is sorted on decreasing maximum value. The algorithm⁹ has a complexity $O(n)$ where n is the number of variables.

Finally we make a remark that will be used later on, in order to shrink variables. Let $U_{i_1}, \dots, U_{i_{ndistinct}}$ be a subset of variables $U_1, \dots, U_n$ such that intervals $\min(U_{i_1}).\max(U_{i_1}), \dots, \min(U_{i_{ndistinct}}).\max(U_{i_{ndistinct}})$ do not pairwise intersect. If at least one variable of $U_1, \dots, U_n$ takes a value that does not belong to the union of intervals $\min(U_{i_1}).\max(U_{i_1}), \dots, \min(U_{i_{ndistinct}}).\max(U_{i_{ndistinct}})$, then the minimum number of distinct values in $U_1, \dots, U_n$ will be strictly greater than the quantity $ndistinct$ returned by the algorithm. This is because we would get $ndistinct+1$ pairwise non-intersecting variables: the “ $ndistinct$ ” $U_{i_1}, \dots, U_{i_{ndistinct}}$ variables, plus the additional variable that we fix. In the example of Figure 1, we can remove from variables $U_1, \dots, U_9$ all values that do not belong to $\min(U_2).\max(U_2) \cup \min(U_7).\max(U_7) \cup \min(U_9).\max(U_9) = \{0, 1, 3, 4, 5\}$, namely $\{2, 6, 7\}$ if we don’t want to have more than three distinct values. But we can also remove all values that do not belong to $\min(U_5).\max(U_5) \cup \min(U_7).\max(U_7) \cup \min(U_9).\max(U_9) = \{1, 2, 3, 4, 5\}$, namely $\{0, 6, 7\}$. We show how to modify the previous algorithm in order to get the values to remove if one wants to avoid having more than $ndistinct$ distinct values. The new algorithm uses two additional arrays $kinf[1..n]$ and $ksup[1..n]$ for recording the lower and upper limits of the intervals of values that we don’t have to remove. These intervals will be called the kernel of $U_1, \dots, U_n$.

```

1  ndist:=1;
2  reinit:=1;
3  i:=1;
4  start_previous_group:=1;
5  WHILE i+1-reinit≤n DO
6    i:=i+1-reinit;
7    IF reinit OR low < min(Ui) THEN low:=min(Ui); ENDIF;
8    IF reinit OR up > max(Ui) THEN up:=max(Ui); ENDIF;
9    reinit:=(low>up);
```

⁹ We did not include the sorting phase of the variables within the algorithm since, in Section 5, we call this algorithm several times on different parts of a given array of variables sorted on their decreasing maximum value.

```

10  IF reinit OR i=n THEN
11    kinf[ndist] := min_item ;
12    ksup[ndist] := max_item ;
13    FOR j:=start_previous_group TO i-reinit DO
14      before:= reinit=0 OR max( $U_j$ ) < min( $U_i$ ) ;
15      IF before
16        AND min( $U_j$ ) > kinf[ndist] THEN kinf[ndist] := min( $U_j$ ) ENDIF;
17      IF max( $U_j$ ) < ksup[ndist] THEN ksup[ndist] := max( $U_j$ ) ENDIF;
18    ENDFOR;
19    start_previous_group:=i;
20  ENDIF;
21  ndist:=ndist+reinit;
22  ENDWHILE;

22  IF ndist>ndistinct THEN FAIL10;
23  ELSE IF ndist=ndistinct THEN
24    adjust minimum values of  $U_1,...,U_n$  to kinf[1];
25    adjust maximum values of  $U_1,...,U_n$  to ksup[ndistinct];
26    FOR j:=1 TO ndistinct-1 DO
27      remove intervals of values ksup[j]+1..kinf[j+1]-1
28      from  $U_1,...,U_n$  ;
29    ENDFOR;
30  ENDIF;

```

The complexity of lines 1 to 21 is still in $O(n)$, while the complexity of lines 22 to 29 is proportional to the number of values we remove from the domain of variables $U_1,...,U_n$. If we run this algorithm on the example of Figure 1, we get three intervals kinf[1]..ksup[1], kinf[2]..ksup[2] and kinf[3]..ksup[3] that respectively correspond to 1..1, 3..3 and 4..5. The lower and upper limits of interval 1..1 were respectively obtained by the minimum value of U_5 (see lines 14,15: U_5 is a variable for which $\max(U_5) < \min(U_6) = 3$) and the maximum value of U_2 (see line 16). From this we deduce that, if we don't want to have more than three distinct values, all variables $U_1,...,U_9$ should be greater or equal than 1, less or equal than 5, and different from 2.

4 Computing a Lower Bound of the $(r+1)^{th}$ Smallest Distinct Value of a Set of Variables

When r is equal to 0 we scan the variables and returns the associated minimum value. When r is greater than 0, we use the following greedy algorithm that

¹⁰ FAIL indicates that the constraint cannot hold and that we therefore exit the procedure; for simplicity reason we omit the FAIL in lines 24, 25 and 27, but it should be understand that adjusting the minimum or the maximum value of a variable, or removing values from a variable could also generate a FAIL.

successively produces the $r+1$ smallest distinct values by starting from the smallest possible value of a set of variables $\{U_1, \dots, U_n\}$. At each step of the algorithm we extract one variable from $\{U_1, \dots, U_n\}$ according to the following priority rule: we select the variable with the smallest minimum value and with the minimum largest value in case of tie (line 6). The key point is that at iteration k we consider the minimum value of all remaining variables to be at least equal to the $(k-1)^{th}$ smallest value $\min$ produced so far (or to $\min_item$ if $k=1$).

```

1  min:= min_item ;
2  SU := {U1, ..., Un} ;
3  k:=1;
4  DO
5    IF k>n THEN BREAK ENDIF;
6    U := a variable of SU with the smallest value for
        maximum(min(U), min), and the smallest value for max(U)
        in case of tie;
7    SU := SU - {U};
8    IF k=1 OR min < max(U) THEN
9      IF k=1 OR min < min(U) THEN min:= min(U)
        ELSE min:= next(min) ENDIF;
10   r:=r-1;
11   ENDIF;
12   k:=k+1;
13   WHILE r≥0;
14   IF r=-1 THEN RETURN min ELSE RETURN max_item ENDIF;

```

Next table shows for $r=6$ and for the set of variables $\{U_1, \dots, U_9\}$ with the respective domain 4..9, 5..6, 0..1, 3..4, 0..1, 0..1, 4..9, 5..6, 5..6 the state of k , U , $\min$ and r just before execution of the statement of line 12. From this we find out that the $(6+1)^{th}$ smallest distinct value is greater or equal than 7.

Table 2. State of the main variables at the different iterations of the algorithm

k	1	2	3	4	5	6	7	8	9
U	0..1	0..1	0..1	3..4	4..9	5..6	5..6	5..6	4..9
min	0	1	1	3	4	5	6	6	7
r	5	4	4	3	2	1	0	0	-1

In order to avoid the rescanning implied by line 6, and to have an overall complexity of $O(n \lg n)$, we rewrite the previous algorithm by using a heap which contain variables $U_1, \dots, U_n$ sorted in increasing order of their maximum.

```

1  let S1, ..., Sn be the variables U1, ..., Un sorted in increasing order
        of minimum value;
2  creates an empty heap;
3  k:=1;
4  DO
5    extract from the heap all variables S for which:
        max(S) < min ∨ max(S) = min;

```



```

6   IF k>n AND empty heap THEN BREAK ENDIF;
7   IF empty heap THEN min:=min( $S_k$ ) ELSE min:=next(min) ENDIF;
8   WHILE k≤n AND min( $S_k$ )=min DO
9       push  $S_k$  on the heap;
10      k:=k+1;
11  ENDWHILE;
12  extract from the heap variable with smallest maximum value;
13  r:=r-1;
14  WHILE r≥0;
15  IF r=-1 THEN RETURN min ELSE RETURN max_item ENDIF;

```

5 Pruning of M

The minimum value of M corresponds to the smallest $(r+1)^{th}$ item that can be generated from the values of variables $V_1, \dots, V_n$. Note that, since all variables that take the same value will have the same rank according to the ordering relation $\mathfrak{R}$, we have to find $r+1$ distinct values. For this purpose we use the last algorithm introduced in Section 4. Note that the previous algorithm will return `max_item` if there is no way to generate $r+1$ distinct values; since this is the biggest possible value, this will fix M to value `max_item`.

When r is equal to 0, the maximum value of M is equal to the smallest maximum value of variables $V_1, \dots, V_n$. When r is greater than 0, the maximum value of M is computed in the following way by the next three methods. We denote `min_nval( $U_1, \dots, U_m$ )` a call to the algorithm that computes a lower bound of the minimum number of distinct values of a set of variables $\{U_1, \dots, U_m\}$ (see first algorithm of Section 3). We sort variables $V_1, \dots, V_n$ in decreasing order on their maximum value and perform the following points in that given order:

- if none of $V_1, \dots, V_n$ can take `max_item` as value, and if there are at least $r+1$ distinct values for variables $V_1, \dots, V_n$ (i.e. `min_nval( $V_1, \dots, V_n$ ) ≥  $r+1$`) then we are sure that the $(r+1)^{th}$ item will be always defined; so we update the maximum value of M to `prev(max_item)`.
- if the maximum value of M is less than `max_item`, we make a binary search (on $V_1, \dots, V_n$ sorted in decreasing order on their maximum value) of the largest suffix for which the minimum number of distinct values is equal to $r+1$; finally, we update the maximum value of M to the maximum value of the variables of the previous largest suffix. This is a valid upper bound for M , since taking a larger value for the smallest $(r+1)^{th}$ distinct value would lead to at least $r+2$ distinct values. Since the linear procedure described in Section 3 is called no more than $\lg n$ times, the overall complexity of this step is $O(n \lg n)$.
- When the largest suffix founded at the previous step contains all variables $V_1, \dots, V_n$ we update the maximum value of M to the maximum value of the kernel of

$V_1, \dots, V_n$. This is the value `ksup[ndist]` computed by the second algorithm of Section 3. This is again a valid upper bound since taking a larger value for M would lead to $r+2$ distinct values: by definition of the kernel (see Section 3), all values that are not in the kernel lead to one additional distinct value.

Let us illustrate the pruning of the maximum value of M on the instance $\text{min_n}(M, 1, \{V_1, \dots, V_9\})$, with $V_1, \dots, V_9$ having respectively the following domains 0..3, 0..1, 1..7, 1..6, 1..4, 3..4, 3..3, 4..6 and 4..5, and M having the domain 0..9. By sorting $V_1, \dots, V_9$ in decreasing order on their maximum value we obtain $V_3, V_4, V_8, V_9, V_5, V_6, V_1, V_7, V_2$. We then use a binary search that starts from interval 1..9 and produces the following sequence of queries:

- `inf=1, sup=9, mid=5; min_nval( $V_5, V_6, V_1, V_7, V_2$ )` returns 2 that is less or equal than $r+1=2$,
- `inf=1, sup=5, mid=3; min_nval( $V_8, V_9, V_5, V_6, V_1, V_7, V_2$ )` returns 3 that is greater than $r+1=2$,
- `inf=4, sup=5, mid=4; min_nval( $V_9, V_5, V_6, V_1, V_7, V_2$ )` returns 3 that is greater than $r+1=2$.

From this, we deduce that the maximum value of M is at most equal to the maximum value of variable V_5 , namely 4.

Finally, since variable M will be equal to one of the variables $V_1, \dots, V_n$ or to value `max_item`, we must remove from M all values different from `max_item`, that do not belong to any variable of $V_1, \dots, V_n$. If only one single variable of $V_1, \dots, V_n$ has some values in common with M , and if M cannot take `max_item` as value, then this variable should be unified¹¹ with M .

6 Pruning of $V_1, \dots, V_n$

Pruning of variables $V_1, \dots, V_n$ is achieved by using the following deduction rules:

- Rule 1: If $n-r-1$ variables are greater than M then the remaining variables are less or equal than M ¹².
- Rule 2: If $M < \text{max_item}$ then we have at least $r+1$ distinct values for the variables of $V_1, \dots, V_n$ that are less or equal than M .
- Rule 3: We have at most $r+1$ distinct values for the variables of $V_1, \dots, V_n$ that are less or equal than M .

¹¹ Some languages such as Prolog for instance offer unification as a basic primitive. If this is not the case then one has to find a way to simulate it. This can be achieved by using equality constraints.

¹² If there are not $r+1$ distinct values among variables $V_1, \dots, V_n$ then variable M takes by definition value `max_item` (see Section 2) and therefore all variables $V_1, \dots, V_n$ are less or equal than M .

- Rule 4: If $M \prec \max_item$ then we have at least r distinct values for the variables of $V_1, \dots, V_n$ that are less than M .
- Rule 5: We have at most r distinct values for the variables of $V_1, \dots, V_n$ that are less than M .

Rules 2 and 4 impose a condition on the minimum number of distinct values, while rules 3 and 5 enforce a restriction on the maximum number of distinct values. In order to implement the previous rules we consider the following subset of variables of $V_1, \dots, V_n$:

- $V_{<}$ is the set of variables V_i that are for sure less than M (i.e. $\max(V_i) < \min(M)$),
- $V_{\leq}$ is the set of variables V_i that are for sure less or equal than M (i.e. $\max(V_i) \leq \min(M)$),
- $V_{>}$ is the set of variables V_i that are for sure greater than M (i.e. $\min(V_i) > \max(M)$),
- $\overline{V_{>}}$ is the set of variables V_i that may be less or equal than M (i.e. $\min(V_i) \leq \max(M)$),
- $\overline{V_{\geq}}$ is the set of variables V_i that may be less than M (i.e. $\min(V_i) < \max(M)$),
- $\overline{V_{<}}$ is the set of variables V_i that may be greater or equal than M (i.e. $\max(V_i) \geq \min(M)$),
- $\overline{V_{\leq}}$ is the set of variables V_i that may be greater than M (i.e. $\max(V_i) > \min(M)$).

$|V_{>}|$ denotes the number of variables in $V_{>}$. We also introduce the four following algorithms that take a subset of variables V of $V_1, \dots, V_n$ and an integer value $vmax$ as arguments, and perform the respective following task:

- $\min_nval(V)$ is a lower bound of the minimum number of distinct values of the variables of V ; it is computed with the first algorithm we have introduced in Section 3,
- $\min_nval_prune(V, vmin)$ removes from variables $V_1, \dots, V_n$ all values less or equal than $vmin$ that do not belong to the kernel of V ; it uses the last algorithm of Section 3,
- $\max_matching(V, vmax)$ is the size of the maximum matching of the following bipartite graph: the two classes of vertices correspond to the variables of V and to the union of values, less or equal than a given limit $vmax$, of the variables of V ; the edges are associated to the fact that a variable of V takes a given value that is less or equal than $vmax$; when we consider only intervals for the variables of V , it can be computed in linear time in the number of variables of V with the algorithm given in [9].
- $\text{matching_prune}(V, vmax)$ removes from the bipartite graph associated to V and $vmax$ all edges that do not belong to any maximum matching (this includes values

which are greater than v_{max}); for this purpose we use the algorithm given in [4] or [8].

We now restate the deduction rules in the following way:

Rule 1: IF $|V_{>}| = n - r - 1$ THEN $\forall V_i \in \overline{V_{>}} : \max(V_i) < \text{next}(\max(M))$

Rule 2: IF $\max(M) < \max_item$ AND $\max_matching(\overline{V_{>}}, \max(M)) < r + 1$ THEN fail
 ELSE IF $\max(M) < \max_item$ AND $\max_matching(\overline{V_{>}}, \max(M)) = r + 1$ THEN
 $\text{matching_prune}(\overline{V_{>}}, \max(M))$

Rule 3: IF $\min_nval(V_{\leq}) > r + 1$ THEN fail
 ELSE IF $\min_nval(V_{\leq}) = r + 1$ THEN $\min_nval_prune(V_{\leq}, \min(M))$

Rule 4: IF $\max(M) < \max_item$ AND $\max_matching(\overline{V_{\geq}}, \text{prev}(\max(M))) < r$ THEN fail
 ELSE IF $\max(M) < \max_item$ AND $\max_matching(\overline{V_{\geq}}, \text{prev}(\max(M))) = r$ THEN
 $\text{matching_prune}(\overline{V_{\geq}}, \text{prev}(\max(M)))$

Rule 5: IF $\min_nval(V_{<}) > r$ THEN fail
 ELSE IF $\min_nval(V_{<}) = r$ THEN $\min_nval_prune(V_{<}, \text{prev}(\min(M)))$

We give several examples of application of the previous deduction rules.

$\min_n(M : 2..3, r : 1, \{V_1 : 0..9, V_2 : 4..9, V_3 : 0..9\}) :$

Rule 1: Since $V_{>} = \{V_2\}$ and $|V_{>}| = n - r - 1 = 3 - 1 - 1 = 1$, we have: $\begin{cases} \max(V_1) \leq \max(M) = 3 \\ \max(V_3) \leq \max(M) = 3 \end{cases}$.

$\min_n(M : 4..6, r : 3, \{V_1 : 3..4, V_2 : 3..4, V_3 : 3..4, V_4 : 6..9, V_5 : 7..9\}) :$

Rule 2: No solution since $\overline{V_{>}} = \{V_1, V_2, V_3, V_4\}$ and $\max_matching(\overline{V_{>}}, 6) = 3 < r + 1 = 4$.

$\min_n(M : 1..2, r : 2, \{V_1 : 0..1, V_2 : 0..3, V_3 : 0..1, V_4 : 3..7\}) :$

Rule 2: Since $\overline{V_{>}} = \{V_1, V_2, V_3\}$ and $\max_matching(\overline{V_{>}}, 2) = 3 = r + 1$, we have: $V_2 = 2$.

$\min_n(M : 6..7, r : 1, \{V_1 : 0..1, V_2 : 1..2, V_3 : 3..4, V_4 : 0..3, V_5 : 4..5, V_6 : 5..6, V_7 : 2..9\}) :$

Rule 3: No solution since $V_{\leq} = \{V_1, V_2, V_3, V_4, V_5, V_6\}$ and $\min_nval(V_{\leq}) = 3 > r + 1 = 2$.
 ($\min_nval(V_{\leq})$ is equal to 3 since intervals $\min(V_1).. \max(V_1)$, $\min(V_3).. \max(V_3)$ and $\min(V_6).. \max(V_6)$ do not pairwise intersect)

$\min_n(M : 6..7, r : 2, \{V_1 : 0..1, V_2 : 1..2, V_3 : 3..4, V_4 : 0..3, V_5 : 4..5, V_6 : 5..6, V_7 : 2..9\}) :$

Rule 3: Since $V_{\leq} = \{V_1, V_2, V_3, V_4, V_5, V_6\}$ and $\min_nval(V_{\leq}) = 3 = r + 1$ and because intervals $\min(V_1).. \max(V_1)$, $\min(V_3).. \max(V_3)$ and $\min(V_6).. \max(V_6)$ do not pairwise intersect, we can remove all values, less or equal than $\min(M) = 6$, that do not belong to $\min(V_1).. \max(V_1) \cup \min(V_3).. \max(V_3) \cup \min(V_6).. \max(V_6) = \{0, 1\} \cup \{3, 4\} \cup \{5, 6\}$; therefore we remove value 2 from V_2 , V_4 and V_7 .

$\min_n(M : 4..6, r : 3, \{V_1 : 1..2, V_2 : 1..2, V_3 : 1..2, V_4 : 6..9, V_5 : 7..9\}) :$

Rule 4: No solution since $\overline{V_{\geq}} = \{V_1, V_2, V_3\}$ and $\max_matching(\overline{V_{\geq}}, 5) = 2 < r = 3$.

$\min_n(M : 4..6, r : 3, \{V_1 : 1..2, V_2 : 1..3, V_3 : 1..2, V_4 : 6..9, V_5 : 7..9\}) :$

Rule 4: Since $\overline{V_{\geq}} = \{V_1, V_2, V_3\}$ and $\max_matching(\overline{V_{\geq}}, 5) = 3 = r$, we have: $V_2 = 3$.

$\min_n(M : 5..6, r : 1, \{V_1 : 0..1, V_2 : 1..2, V_3 : 3..4, V_4 : 5..9, V_5 : 0..9\}) :$

Rule 5: No solution since $V_{<} = \{V_1, V_2, V_3\}$ and $\min_nval(V_{<}) = 2 > r = 1$ ($\min_nval(V_{<})$ is equal to 2 since intervals $\min(V_1)..max(V_1)$ and $\min(V_3)..max(V_3)$ are disjoint).

$\min_n(M : 5..6, r : 2, \{V_1 : 0..1, V_2 : 1..2, V_3 : 3..4, V_4 : 5..9, V_5 : 0..9\}) :$

Rule 5: Since $V_{<} = \{V_1, V_2, V_3\}$ and $\min_nval(V_{<}) = 2 = r$ and because the two intervals $\min(V_1)..max(V_1)$ and $\min(V_3)..max(V_3)$ are disjoint, we can remove all values, strictly less than $\min(M) = 5$, that do not belong to $\min(V_1)..max(V_1) \cup \min(V_3)..max(V_3) = \{0,1\} \cup \{3,4\}$; therefore we remove value 2 from V_2 and V_5 . In addition, since the two intervals $\min(V_2)..max(V_2)$ and $\min(V_3)..max(V_3)$ are disjoint, we can also remove value 0 from V_1 and V_5 .

7 Defining the *minimum* Family Constraint in Terms of *Forbidden Regions*

In [2] we have introduced a generic geometrical pruning technique that is based on the aggregation of several constraints that share some variables in common. In order to be used, this technique requires defining the set of *forbidden regions* associated to a constraint. We first recall what a *forbidden region* is, and then show how to use the pruning rules introduced in the previous sections in order to define the *minimum* family constraint in terms of *forbidden regions*. This corresponds to another more indirect way of interpreting and using the pruning rules.

Definition *forbidden region according to a given constraint $C(V_1, \dots, V_n)$ and 2 given variables*

Let C be a constraint that specifies a condition on variables $V_1, \dots, V_n$. A *forbidden region* F according to constraint C and according to two given distinct variables V_i and V_j ($1 \leq i < j \leq n$) is defined by two intervals $\inf_{F, V_i} .. \sup_{F, V_i}$ and $\inf_{F, V_j} .. \sup_{F, V_j}$ such that: $\forall v_i, v_j / v_i \in \inf_{F, V_i} .. \sup_{F, V_i} \wedge v_j \in \inf_{F, V_j} .. \sup_{F, V_j} : C(V_1, \dots, V_n)$ with the assignment $V_i = v_i$ and $V_j = v_j$ has no solution.

For each rule there are two different ways of using it in order to define *forbidden regions*:

- a first way consists to keep the rule as it is and to construct the *forbidden regions* associated to the constraint that is enforced in the “then” part of the rule. Since we generally¹³ impose inequality or disequality constraints this is straightforward.
- a second more indirect way is as follows. Typically, the “if” part of all deduction rules checks that the cardinality of a given set of variables SET is equal to some given fixed number fix . Getting more information about *forbidden regions* for two variables requires using the rules in an anticipated mode where we trigger the rule one step earlier. For achieving this, we rewrite the cardinality check by replacing fix by $fix-1$ or $fix+1$, depending on the fact whether the number of variables of SET increases or decreases over time. For instance, the cardinality of $V_<$, $V_≤$ and $V_>$ increase while the cardinality of $\overline{V_>}$ and $\overline{V_≤}$ decrease when the domains of $V_1, \dots, V_n$ and M are reduced. We then try to combine the premise and conclusion of the rule in order to get a forbidden condition involving two variables.

We now restate the deduction rules in terms of *forbidden regions*. We only consider rules 1 to 3, since rule 4 is similar to rule 2, and rule 5 is similar to rule 3. Parts (A) and (B) of Figure 2 will respectively correspond to the first and second way of defining *forbidden regions*.

Translation of rule 1

Part (A) of Figure 2 can be interpreted as follows: if $n-r-1$ variables are for sure greater than M then we forbid for all variables V_i that are not for sure greater than M to be greater than M ; there is a *forbidden region* for all pairs of variables (M, V_i) such that $V_i \in \overline{V_>}$. Part (B) can be interpreted as follows: if $n-r-2$ variables are greater than M then, for any pair of variables V_i, V_j that are not for sure greater than M , both variables V_i and V_j should not be simultaneously greater than M .

Fig. 2. Forbidden regions associated to rule 1

¹³ Except for rules 3 and 5 where we restrict the maximum number of distinct values.

Translation of rule 2

Figure 3 can be interpreted as follows: if the number of variables that may be less or equal than M is equal to $r+1$, then all the variables that may be less or equal than M should all be pairwise different. For such pairs V_i, V_j of variables we forbid both variables to take the same value.

IF $\max(M) \prec \max_item$ AND $|\overline{V_{\geq}}| = r+1$:

Fig. 3. Forbidden regions associated to rule 2

Translation of rule 3

Figure 4 can be interpreted as follows: if the minimum number of distinct values less or equal than M is just equal to the maximum possible limit $r+1$, then we forbid for all pairs of variables to take two distinct values belonging to the same kernel¹⁴ interval¹⁵. These values correspond to the 8 dark gray squares. The other values are already removed by the procedure $\min_nval_prune(V_{\leq}, \min(M))$ of rule 3. Figure 4 assumes the kernel to be constituted from intervals 4..6 and 9..10.

¹⁴ For the notion of kernel refer to Section 3.

¹⁵ Since the following property holds: if the number of intervals of the kernel is equal to the maximum number of distinct values to produce then for each interval of the kernel only one single value has to be selected.

IF $\min_nval(V_{\leq}) = r + 1$:

Fig. 4. Forbidden regions associated to rule 3

8 The Number of Distinct Values Constraint

The *number of distinct values* constraint has the form $nvalue(D, \{V_1, \dots, V_n\})$ where D is a domain variable and $\{V_1, \dots, V_n\}$ is a collection of variables. The constraint holds if D is the number of distinct values taken by the variables $V_1, \dots, V_n$. This constraint was introduced in [7, page 339] and in [1, page 37], but a propagation algorithm for this constraint was not given. The *nvalue* constraint generalizes several more simple constraints like the *alldifferent* and the *notallequal*¹⁶ constraints. The purpose of this section is to show how to reduce the minimum and maximum values of D and how to shrink the domains of $V_1, \dots, V_n$:

- since the minimum value of D is the minimum number of distinct values that will be taken by variables $V_1, \dots, V_n$, one can sort variables $V_1, \dots, V_n$ on increasing minimum value and use the first algorithm described in Section 3 in order to get a lower bound of the minimum number of distinct values. Then the minimum of D will be adjusted to the previous computed value.
- since the maximum value of D is the maximum number of distinct values that can be taken by variables $V_1, \dots, V_n$, one can use a maximum matching algorithm on the following bipartite graph: the two classes of vertices of the graph are the variables $V_1, \dots, V_n$ and the values that can be taken by the previous variables; the edges are associated to the fact that a variable of $V_1, \dots, V_n$ takes a given value. The maximum value of D will be adjusted to the size of the maximum matching of the previous bipartite graph.
- the following rules, respectively similar to rules 2 and 3 of Section 6, are used in order to prune the domain of variables $V_1, \dots, V_n$:

¹⁶ The *notallequal*($\{V_1, \dots, V_n\}$) constraint holds if the variables $V_1, \dots, V_n$ are not all equal.

$$\begin{aligned} & \text{IF } \text{max_matching}(V_1, \dots, V_n, \text{MAXINT}) = \min(D) \text{ THEN} \\ & \qquad \qquad \qquad \text{matching_prune}(V_1, \dots, V_n, \text{MAXINT}), \\ & \text{IF } \text{min_nval}(V_1, \dots, V_n) = \max(D) \text{ THEN } \text{min_nval_prune}(V_1, \dots, V_n, \text{MAXINT}). \end{aligned}$$

The first rule enforces to have at least $\min(D)$ distinct values, while the second rule propagates in order to have at most $\max(D)$ distinct values.

Finally, we point out that one can generalize the *number of distinct values* constraint to the *number of distinct values* constraint family by requiring to count the number of distinct equivalence classes taken by the values of variables $V_1, \dots, V_n$ according to a given equivalence relation.

9 Conclusion

We have presented generic propagation rules for the *minimum* and *nvalue* constraints families and two algorithms that respectively compute a lower bound for the minimum number of distinct values and for the $(r+1)^{th}$ smallest distinct value. These algorithms produce a tight lower bound when each domain consists of one single interval of consecutive values. However there should be room for improving these algorithms in order to try to consider holes in the domains of variables. One should also provide for small values of r an algorithm for computing the r^{th} smallest distinct value of a set of intervals for which the complexity depends of r . We did not address any incremental concern since it would involve other issues like maintaining a list of domain variables sorted on their minimum, or like regrouping all propagation rules together in order to factorize common parts. Finally one original contribution of this paper is to show how to characterize a global constraint in terms of *forbidden regions* that can be used by the sweep algorithm introduced in [2]. Deriving global *forbidden regions* should also be systematically investigated for other families of global constraints.

Acknowledgements

Thanks to Mats Carlsson and Per Mildner for useful comments on an earlier draft of this report.

References

1. Beldiceanu, N.: Global Constraints as Graph Properties on Structured Network of Elementary Constraints of the Same Type. SICS Technical Report T2000/01, (2000).
2. Beldiceanu, N.: Sweep as a generic pruning technique. SICS Technical Report T2000/08, (2000).
3. Cormen, T. H., Leiserson, C. E., Rivest R. L.: *Introduction to Algorithms*. The MIT Press, (1990).
4. Costa, M-C.: Persistency in maximum cardinality bipartite matchings. *Operation Research Letters* 15, 143-149, (1994).
5. Damaschke, P., Müller, H., Kratsch, D.: Domination in convex and chordal bipartite graphs. *Information Processing Letters* 36, 231-236, (1990).
6. Garey, M. R., Johnson, D. S.: *Computers and intractability. A guide to the Theory of NP-Completeness*. W. H. Freeman and Company, (1979).
7. Pachet, F., Roy, P.: Automatic Generation of Music Programs. In *Principles and Practice of Constraint Programming - CP99, 5th International Conference*, Alexandria, Virginia, USA, (October 11-14, 1999), Proceedings. Lecture Notes in Computer Science, Vol. 1713, Springer, (1999).
8. 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).
9. Steiner, G., Yeomans, J.S.: A Linear Time Algorithm for Maximum Matchings in Convex, Bipartite Graph. In *Computers Math. Applic.*, Vol. 31, No. 12, pp.91-96, (1996).