# `PrefixCCFWC`: technical report

Victor Lecomte

September 3, 2015

**Abstract**

In scheduling, it may be useful to specify cardinality constraints on certain prefixes of an array of variables. For example, if only one item can be produced each day, and you have to deliver 3 units of product $A$ after 7 days, you will want to impose that there be at least 3 occurences of $A$ among the production variables for the first 7 days.

This constraint allows you to apply many such constraints on several values without the overhead of creating a GCC for each of them, and with some additional pruning.

## Contents

## 1 Problem statement

We are given an array of integer variables and a number of constraints of the form:

There should be at least/most $b$ occurrences of value $v$ among the $i$ first variables.

In scheduling this will mostly be lower bounds coming from quantities to be produced at a certain date, but there could also be upper bounds coming from storage limitations.

For example let's consider four variables with values either `A` or `B`, and the following constraints:

— there should be at least one `B` in the first two variables ($b = 1$, $v = \texttt{B}$, $i = 2$);

— there should be at most two `A`s in all the variables ($b = 2$, $v = \texttt{A}$, $i = 4$).

Then the following results would be valid:

— `A B A B`

— `B B B B`

While the following results would be invalid:

— `A A B B` (no `B` in the first two variables)

— `B A A A` (too many `A`s)

# 2 The algorithm

The algorithm consists of two main parts:

1. The deduction and filtering of the bounds, where we analyze the bounds given to filter out the redundant ones and add additional ones when possible.
2. The propagation, which takes the bounds obtained in the first part and applies pruning identical to regular forward-checking GCC, but in an efficient unified manner.

We will start by explaining the *bound deduction* (2.1) and *bound filtering* (2.2) steps, then we will introduce the concept of *critical values* (2.3) and how it is the key in unifying the bound checking, and finally we will go on to the *merging and pruning process* (2.4), the main propagation mechanism.

## 2.1 Bound deduction

The deduction step aims to obtain better bounds for the occurrences of a value than those given in the input, based on two factors:

— the values of the bounds on the same value for other prefixes, “inter-prefix”;

— the values of the opposite bounds for the same prefix on other values, “inter-value”.

### 2.1.1 Inter-prefix deduction

The first factor is based on these four types of deduction:

— if there are at least three `As` in the first five variables, there are also at least three `As` in the first six variables (I), and at least *two* in the first four (because we are only possibly removing one) (II);

— if there are at most three `As` in the first five variables, there are also at most three `As` in the first four variables (III), and at most *four* in the first six (because we are only possibly adding one) (IV).

In practice, those deductions can be made by maintaining an array of the best known bounds for each prefix and then traversing once forwards for deductions (I) and (IV) and once backwards for deductions (II) and (III).

Here is a pseudocode, for a certain value:

```
for i in 1 to (numberOfVariables-1):
    lower(i) = max(lower(i), lower(i-1))
    upper(i) = min(upper(i), upper(i-1) + 1)
for i in (numberOfVariables-2) to 0:
    lower(i) = max(lower(i), lower(i+1) - 1)
    upper(i) = min(upper(i), upper(i+1))
```

### 2.1.2 Inter-value deduction

The second factor is based on these two types of deduction:

— in the first five variables, if the minimal number of occurrences for all the other values is three, then we can decrease the maximal number of occurrences for this value to two (since there are at most for five occurences in total);

— in the first five variables, if the maximal number of occurrences for all the other values is three, then we can increase the minimal number of occurrences for this value to two (since there are at least five occurrences in total).

In practice, those deductions can be made for each prefix and for each value by computing the sums of all the lower (resp. upper) bounds for all of the other values and setting the upper (resp. lower) bound of this value to the length of the prefix minus that sum (if this results in a stronger bound).

Here is a pseudocode, for a certain prefix:

```
for v in values:
    lower(v) = max(lower(v), sizeOfPrefix - (sum(upper) - upper(v)))
    upper(v) = min(upper(v), sizeOfPrefix - (sum(lower) - lower(v)))
```

## 2.2 Bound filtering

Once the algorithm has deduced the best bounds it could, one would like to filter them and only keep those that add information. This step sounds a lot like applying the inter-prefix deduction in reverse: instead of creating new bounds by deducing them from the next or previous prefix, we will remove them if they can be deduced in that way.

In practice, we will traverse the prefixes from left to right, and:

1. only add a bound if it gives more information than the previous bound;
2. remove the previous bounds that can be deduced from the bound we're adding.

That second point is similar in spirit with the "monotone chain" algorithm for convex hulls: we're trying to find a minimal set of bounds (resp. points) and to ensure it's minimal, when adding a new bound (resp. point), we remove the previous bounds (resp. points) as long as they're being made useless by the new one.

For lower bounds, we base ourselves on the inter-prefix deduction laws (see 2.1) in this way:

— a bound only adds information if it is higher than the previous one (I);

— a previous bound has to be removed if it is lower or equal to the new bound *minus their distance* (II).

For example, if we have a lower bound of one on the first three variables, we can add a lower bound of three on the first five variables, because it adds informations. However, we will then have to remove the previous constraint, because that lower bound of one can be directly deduced from the one we're adding.

In general, we can visually check if a lower bound is stronger than another by "extending" it, decreasing constantly on the left and staying constant on the right, like this: **TODO: insert diagram**

Here is a pseudocode of the filtering:

```
for cur in lowerBounds:
    if cur.bound > last.bound:
        while last.bound <= cur.bound - (cur.index - last.index):
            removeLast()
        add(cur)
```

The situation is similar for upper bounds:

— a bound only adds information if it is lower than the previous one *plus their distance* (IV);

— a previous bound has to be removed if it is higher or equal to the new bound (III).

The “extension” works in the same way: **TODO: insert diagram**

Here is a pseudocode of the filtering:

```
for cur in upperBounds:
    if cur.bound < last.bound + (cur.index - last.index):
        while last.bound >= cur.bound
            removeLast()
        add(cur)
```

### 2.3 The concept of critical values

**TODO**

### 2.4 Merging and pruning

**TODO**

## 3 Complexity

**TODO**

## 4 Conclusion and use cases

**TODO**