A fast algorithm for the bound consistency of alldiff constraints

Jean-Francois Puget
ILOG
9 Av. verdun
94253 Gentilly FRANCE
puget@ilog.fr

Abstract

Some n-ary constraints such as the alldiff constraints arise naturally in real life constraint satisfaction problems (CSP). General purpose filtering algorithms could be applied to such constraints. By taking the semantics of the constraint into account, it is possible to design more efficient filtering algorithms. When the domains of the variables are totally ordered (e.g. all values are integers), then filtering based on bound consistency may be very useful. We present in this paper a filtering algorithm for the alldiff constraint based on bound consistency whose running time complexity is very low. More precisely, for a constraint involving n variables, the time complexity of the algorithm is $O(nlog(n))$ which improves previously published results. The implementation of this algorithm is discussed, and we give some experimental results that prove its practical utility.

1. Introduction

Constraint programming systems are now routinely used to solve complex combinatorial problems in a wide variety of industries. These systems use filtering algorithms based on arc-consistency or bound consistency as subroutines. In real life CSP, n-ary constraints such as the alldiff constraint arise naturally. Although general purpose algorithms could be used, more efficient algorithms have been devised in the past years. These algorithm exploit the mathematical structure of the constraints. For instance Regin [5] proposed to apply graph theory for filtering the alldiff constraint.

In scheduling or time tabling problems, variables representing starting time of activities take their values in an ordered domain: the set of dates, usually represented by a number. In such problems, filtering based on the notion of bound consistency are very useful. In that case, domains are represented by intervals, and the purpose of filtering is to tighten the bounds of these intervals.

The purpose of this paper is to propose a new bound consistency algorithm for one of the most widely used constraint, namely the alldiff constraint. This constraint involves a set of variables x_i, and simply states that the x_i are pairwise different. In other words, the same value cannot be assigned to two variables x_i and x_j, for any pair i,j. This constraint arises naturally in a wide variety of problems, ranging from puzzles (the n queens problem) to assignment problems, scheduling problems and time tabling problems.

More complex examples will be presented later on, but for the sake of clarity, let us consider a very simple time tabling problem, where a set of speeches must be scheduled during one day. Each speech lasts exactly one hour including questions, and only one conference room is available. Moreover, each speaker has other commitments, hence each speaker can only assist a fraction of the day, defined by an earliest and a latest possible time slot. Table 1 gives a particular instance of the problem.

Speaker	min	max
John	3	6
Mary	3	4
Greg	2	5
Susan	2	4
Marc	3	4
Helen	1	6

Table 1. A time tabling problem with 6 time slots.

This problem can easily be encoded as a CSP. We create one variable per speaker, whose value will be the period where he speaks. The initial domains of the variables are the availability interval for the speakers. Since two speeches cannot be held at the same time in the same conference room, the period for two different speakers must be different. The problem can thus be encoded as follows:

$x_1 \in [3,6], x_2 \in [3,4], x_3 \in [2,5], x_4 \in [2,4], x_5 \in [3,4], x_6 \in [1,6], alldiff(x_1,x_2,x_3,x_4,x_5,x_6)$

In this particular example, our algorithm deduces that in all solutions to this problem, x_1 must be assigned to 6, x_2 to 3 or 4, x_3 to 5, x_4 to 2, x_5 to 3 or 4, and x_6 to 1. Hence, the organizers of this event can tell John he will speak at 6, and that Mary can only chose between 3 and 4, for instance. A more realistic time tabling problem involves more than one constraint of course, but this example is sufficient for explaining our algorithm.

A number of filtering algorithms have been proposed for the alldiff constraint. The simplest approach is to consider the alldiff constraint as a set of $n(n+1)/2$ binary constraints: $\forall i < j,\ x_i \neq x_j$

It can easily be shown that applying arc consistency to this set of binary constraints amounts to apply the following inference rule: if x_i is assigned to a given value a, remove a from the domains of the other variables. This filtering is strong enough to solve easy problems such as the nqueens problem (see section 6). However, this filtering does no deductions in our time tabling example. Moreover, this simple filtering algorithm does not even check the satisfiability of the constraint. For instance, it does not reduce any domains in the following inconsistent problem.

$y_1 \in [1,2],\ y_2 \in [1,2],\ y_3 \in [1,2], alldiff(y_1, y_2, y_3)$

Following that remark, several researchers have proposed global filtering algorithms for the alldiff constraint. Regin proposed a graph theoretic approach the computes the arc consistency for the alldiff constraint in $O(n^{2.5})$ [5]. Leconte has proposed an algorithm that runs in $O(n^2)$[3]. This algorithm computes a consistency stronger than bound consistency, but weaker than arc-consistency. Both algorithms deduce the unfeasibility of the previous example, and make the correct deductions in our time tabling example.

We propose an algorithm that computes the bound consistency for the alldiff constraint running in $O(nlog(n))$ time, which improves the $O(n^2)$ complexity of Leconte's algorithm.

When the domains of all the variables appearing in an alldiff constraint are a subset of the interval [1, n], we say that we have a permutation constraint. Bleuzen Guernalec and Colmerauer [1] have recently published an $O(nlog(n))$ algorithm for computing the bound consistency of the permutation constraint. Their algorithm however is not applicable to the general case of the alldiff constraint, contrarily to our work.

The rest of this paper is organized as follows. Section 2 contains a formalization of bound consistency, and the main theoretical result we use, namely Hall's theorem, is introduced. Section 3 presents the derivation of a simple $O(n^3)$ bound consistency algorithm. Using a mathematical property of the alldiff constraint, we derive an $O(n^2)$ algorithm in section 4. Section 5 shows how the previous algorithm can be modified in order to run in $O(nlog(n))$, which is the main contribution of the paper. Section 6 discusses the implementation of this algorithm, and provides experimental results that show its practical usefulness. We conclude by discussing some future directions of research.

2. Theoretical analysis

We will adopt the standard notations of constraint satisfaction problems (CSP). A CSP is defined by a set of variables x_i, a domain $\mathcal{D}$ and a set of constraints $\mathcal{C}$. A finite subset of $\mathcal{D}$ is associated to each variable x_i. This set is called the domain of the variable, and is noted $dom(x_i)$. A constraint c on the variables x_i is defined by a subset of the cartesian product $dom(x_1) \times ... \times dom(x_n)$, representing the set of admissible tuples. Finding a solution to a CSP amounts to select one value for each variable in its domain such that all constraints hold.

We will further assume that $\mathcal{D}$ is totally ordered. For instance D can represent the set of integers. In such a case, we define for each variable x_i its minimum value $min(x_i)$ and its maximum value $max(x_i)$. These values are called *the bounds* of the variable.

We will use the following definition for bound consistency: the constraint is bound consistent, if given any variable, each of its bound can be extended to a tuple satisfying the constraint:

Bound consistency A constraint $c(x_1, x_2, ..., x_n)$ is bound consistent iff for each variable x_i : $\forall a_i \in \{min(x_i), max(x_i)\},\ \forall j \neq i,\ \exists a_j \in [min(x_j), max(x_j)], c(a_1, a_2, ..., a_n)$

For instance, our time tabling example is not bound consistent. Indeed, the min for x_1 is 1, and no solutions exists for $x_1 = 1$.

The following domains are bound consistent for the timetable example:

$x_1 = 6, x_2 \in [3,4], x_3 = 5, x_4 = 3,\ x_5 \in [3,4], x_6 = 1$

The definition above can be used to derive a bound consistency algorithm. However, the complexity of such an algorithm would be exponential in the number of variables appearing in the constraint.

A much faster algorithm is possible using some properties of the constraint we want to filter. Consider any set K of variables. We note by $\#(K)$ the cardinality of K, and $dom(K)$ the union of the domains of the variables in K: $dom(K) = \cup_{x \in K} dom(x)$. Since each variable in K will take one value, we need at least $\#(K)$ values for all the variables in K. In other words, if $\#(K) > \#(dom(K))$, then no solution can be found where all variables in K are assigned to different values. P. Hall [2] proved that this was a necessary and sufficient condition for the existence of a solution. The following corollary of his theorem can be stated in our setting.

Corollary to Hall's theorem: The constraint $alldiff(x_1, \ldots, x_n)$ has a solution if and only if there is no subset K $\subseteq \{x_1, \ldots, x_n\}$ such that $\#(K) > \#(dom(K))$.

If we look back at example 2, consider the set K $= \{y_1, y_2, y_3\}$. We have that dom(K)={1,2}, hence $\#(K) > \#(dom(K))$, which proves that the problem has no solution.

Good filtering algorithms can be derived from Hall theorem. The main idea is the following. If there exists a set K such that $\#(K) = \#(dom(K))$, then we know that any assignment of the variables in K will use all the values in $dom(K)$. Hence these values are not possible for the variables not in K.

For instance, consider the set K $= \{x_2, x_4, x_5\}$ in our time tabling example. The domains of the variables in K are $[3,4], [2,4], [3,4]$, hence dom(K) = {2,3,4}. Then $\#(K) = \#dom(K)$, which implies that the values $\{2,3,4\}$ are not possible for the variables x_1, x_3 and x_6. Hence x_3 must be assigned to 5, and x_1 must be at least 5 for instance. Considering the set K'=$\{x_2, x_5\}$, we deduce that the values $\{3,4\}$ are not possible for x_1, x_3, x_4, x_6, hence x_4 is assigned to 2.

As we are mainly interested in intervals, let us introduce the following.

Definition: Hall Interval Given a constraint $alldiff(x_1, \ldots, x_n)$, and an interval I, let $vars(I)$ be the set of variables x_i such that $dom(x_i) \subseteq I$. We say that I is a *Hall interval* iff $\#(I) = \#(vars(I))$

In our time tabling example, the interval $[3,4]$ is a Hall interval, as it contains two variables x_2 and x_5.

Proposition 2: Given n variables, the number of Hall intervals can be at least n^2.

Proof: consider the following example:

$\forall i,\ 0 \le i \le n,\ dom(x_i) = [i-n, 0]\ \forall i,\ n < i \le 2n,\ dom(x_i) = [0, i-n]$

Then, any interval $I_{i,j} = [i-n, j-n]$ is a Hall interval. Indeed,$I_{i,j}$ contains the domains of all the variables $x_i, x_i+1, .., x_j$, i. e. it contains $j-i+1$ variables. Its width is also $j-i+1$.

We can state the following result.

Proposition 3: The constraint $alldiff(x_1, \ldots, x_n)$ where no $dom(x_i)$ is empty is bound-consistent iff, for each interval I, $\#(vars(I)) \le \#(I)$, and for each Hall interval I, $dom(x_i) \subseteq I$ or $\{min(x_i), max(x_i)\} \cap I = \emptyset$.

Proof: Let I be a Hall interval, and x_i a variable s.t. $dom(x_i)$ is not included in I. Suppose the constraint is bound consistent. By definition the constraint where $dom(x_i)$ is replaced by $min(x_i)$ has a solution. Applying Hall theorem to the set $vars(I)$ results immediately in $min(x_i) \notin I$. For the same reason, $max(x_i) \notin I$. Conversely, suppose that $min(x_i) \in I$ and the $dom(x_i)$ is not included in I. We have that $\#(I) = \#(vars(I))$. If $dom(x_i)$ is replaced by $min(x_i)$, $vars(I)$ is augmented with x_i, resulting in $\#(I) < \#(vars(I))$. From Hall theorem, this means that the new constraint has no solution. A similar reasoning with $max(x_i)$ concludes the proof.

Computing bound consistent domains can in fact be done in two passes. The algorithm that compute new min is applied twice: first to the original problem, resulting into new min bounds, second to the problem where variables are replaced by their inverse, deducing max bounds.

For instance, computing the max of all the variables in the time tabling example can be done by computing the min bounds of the following problem, obtained by replacing each x_i by its inverse z_i:

$\forall i,\ x_i = -z_i,\ z_1 \in [-6,-3], z_2, z_5 \in [-4,-3], z_3 \in [-5,-2], z_4 \in [-4,-2], , z_5 \in [-4,-3], z_6 \in [-6,-1],\ alldiff(z_1, z_2, z_3, z_4, z_5, z_6)$

From this, our algorithm will compute the following new min for the variables z_i: $z_4 \ge -2,\ z_5 \ge -3,\ z_6 \ge -1$. This translates into the following new max for the variables x_i: $x_4 \le 2,\ x_5 \le 3,\ x_6 \le 1$.

From now on, we will only consider the problem of updating the min bounds.

3. A $O(n^3)$ bound consistency algorithm

Using the results of the preceding section, a naive bound consistency algorithm can easily be devised. For all *min* ranging over minimal values of all variables, and for all *max* ranging over maximal values of all variables, consider the interval $I = [min, max]$. If $\#(I) < \#(vars(I))$, there is no solution. If I is a Hall interval, update the bounds that have to be changed.

At each loop, the algorithm treats one variable $x[i]$. The algorithm also maintains for each variable $x[j]$ s.t. $j < i$ a number $u^i[j]$ defined as follows:

$u^i[j] = min[j] + \#(\{k | k < i, min[k] \ge min[j]\}) - 1$

The algorithm computes the u^i numbers incrementally using the following relation:

$u^i[j] = u^{i-1}[j] + Bool(min[i] \ge min[j])$

where $Bool(exp)$ is equal to 1 if exp is true.

Proposition 4: : With the above notation, if $u^i[j] = max[i]$ then the interval $I = [min[j], max[i]]$ is a Hall interval.

Proof: The width of I is $\#(I) = max[i] - min[j] + 1$. The number of variables included in I is

$\#(\{k | max[k] \le max[i], min[k] \ge min[j]\})$

which is greater than or equal to

$\#(\{k | k \le i, min[k] \ge min[j]\})$

which is equal to $u[j] + 1 - min[j]$. If $u[j] = max[i]$ then the above number is equal to $\#(I)$ which concludes the proof.

```
% x is an array containing the variables
% u, min and max are arrays of integers
begin
    SORT(x) %in ascending max order
    for i=1 to n do
        min[i] = min(x[i])
        max[i] = max(x[i])
    for i=1 to n do
        INSERT(i)
end

INSERT(i)
  u[i] ← min[i]
  for j=1 to i-1 do
      if min[j] < min[i] then
          u[j] ← u[j] + 1
          if u[j] > max[i] then Failure
          if u[j] = max[i] then
              INCRMIN(min[j],max[i],i)
      else  u[i] ← u[i] + 1
  if u[i] > max[i] then Failure
  if u[i] = max[i] then  INCRMIN(min[i],max[i],i)

INCRMIN(a,b,i) % [a,b] is a Hall interval
  for j=i+1 to n do
      if min[j] ≥ a then post x[j] ≥ b + 1
```

Algorithm 1: $O(n^3)$ filtering

For each such Hall interval, the algorithm updates the variables $x[j]$ with $j > i$.

In order to obtain what Leconte's algorithm computes, it is sufficient to change the function INCRMIN(a,b,i) : remove $[a, b]$ from the domains of the variables not included in $[a, b]$. Clearly, this is stronger than bound consistency. Note also that Leconte's is more clever than this one, as it runs in $O(n^2)$.

Let's see how this algorithm behaves on the time tabling example. The first step is to sort the variables in ascending order of maximum, yielding $x[1] = x_2, x[2] = x_4, x[3] = x_5, x[4] = x_3, x[5] = x_1, x[6] = x_6$ Then the algorithm loops over these variables as follows.

INSERT(*1*)
max[1] ← $max(x_2) = 4$
min[1] ← $min(x_2) = 3$
u[1] ← 3
INSERT(*2*)
max[2] ← $max(x_4) = 4$
min[2] ← $min(x_4) = 2$
u[2] ← 2
min[1] ≥ min[2] hence u[2] ← 3
INSERT(*3*)
max[3] ← $max(x_5) = 4$
min[3] = ← $min(x_5) = 3$
u[3] ← 3
min[1] ≥ min[3] hence u[3] ← 4
Since u[3] = max[3], calls INCRMIN(*3*,*4*,*3*)
which posts $x_1 \geq 5$
min[2] < min[3] hence u[2] ← 4
Since u[2] = max[3], calls INCRMIN(*2*,*4*,*3*)
which posts $x_1 \geq 5$
and $x_3 \geq 5$
INSERT(*4*)
max[4] ← 5
u[4] ← 2
min[1] ≥ min[4] hence u[4] ← 3
min[2] ≥ min[4] hence u[4] ← 4
min[3] ≥ min[4] hence u[4] ← 5
since u[4]=max[4], call INCRMIN(*2*,*5*,*4*)
which posts $x_1 \geq 6$
Insert5 and Insert6
no more calls to INCRMIN .

4. A $O(n^2)$ bound consistency algorithm

Note that after proposition 2, any algorithm that loops over all Hall intervals has a complexity of at least $O(n^2 \times t(update))$.

It is in fact possible to update the minimum of all variables without examining all Hall intervals. Let's look again at our time tabling example. During the execution of INSERT(*3*), the algorithm discovers two Hall intervals, [2,4] and [3,4] corresponding to the sets of variables $\{x_2, x_5\}$, and $\{x_2, x_4, x_5\}$. We can observe that the updates due to the smaller one ($x_1 \geq 5$) are contained in the updates of the largest one ($x_1 \geq 5$ and $x_3 \geq 5$). This is formalized as follows.

Proposition 5: With the notations of algorithm 1, if $[a, b]$ and $[a', b]$ are two Hall intervals such that $a < a'$, then INCRMIN(a',b,i) does not need to be called.

Proof: The variables updated by the call to INCRMIN(a',b,i) are the variables $x[j]$ such that $i + 1 \leq j \leq n, min[j] \geq a'$. The variables updated by the call to INCRMIN(a,b,i) are the variables $x[j]$ such that $i + 1 \leq j \leq n, min[j] \geq a$, which contains the previous set, since $a' \geq a$.

The revised version of the INSERT function presented in algorithm 2 computes the largest Hall interval ending with $max[i]$ before calling the function INCRMIN . The algorithm obtained by replacing INSERT by INSERT2 runs in $O(n^2)$ time. Indeed, The function INSERT2 is called n times. In each call, the variables with index j smaller than i are visited once, whereas the variables with index j greater than i are visited at most once, when INCRMIN is called.

The behavior of algorithm 2 on the time tabling example is the same as for algorithm 1, except that INCRMIN is called at most once per call to INSERT2 in the main loop.

```
INSERT2(i)
  u[i] ← min[i]
  bestMin ← n + 1
  for j=1 to i-1 do
  |   if min[j] < min[i] then
  |   |   u[j] ← u[j] + 1
  |   |   if u[j] > max[i] then Failure
  |   |   if u[j] = max[i] and min[j] < bestMin then
  |   |   └  bestMin ← min[j]
  |   └
  └   else u[i] ← u[i] + 1
  if u[j] > max[i] then Failure
  if u[i] = max[i] and min[i] < bestMin then
└     bestMin ← min[i]
  if bestIndex ≤ n then
└     INCRMIN(bestMin,max[i], i)
```

Algorithm 2: $O(n^2)$ filtering

5. A $O(nLog(n))$ algorithm

The previous algorithm can still be improved. The main loop of the algorithm is unchanged, i.e. the variables are processed in ascending order of max, but the internals are changed.

```
% x is an array containing the variables
% u, rank, min and max are arrays of integers
begin
|  SORT(x) % ascending max
|  fill in min and max
|  RANK(x)
|  for i=1 to n do
|  |  EXTRACT(x[i])
|  └  INSERT3(x[i])
end
```

Algorithm 3: $O(nlog(n))$ filtering

The structure of the algorithm is similar to that of algorithm 2. it uses two main functions, INSERT3 and INCRMIN3 that are revised versions of INSERT2 and INCRMIN respectively.

After sorting the array x and filling the arrays *min* and *max* as before, the algorithm computes the *rank* of each variable, in the function RANK . This function sorts a copy of x in ascending order of min, then associates to each variable its rank in that ordering. The role of this rank will be explained later.

Then, we find the same main loop. The variables are processed in ascending order of max. For each of them, INSERT3 function is called. As in algorithm 2, INSERT3(i) calls INCRMIN3 for the largest Hall interval ending with $max[i]$ if any. N is a balanced binary tree whose leaves contain all the variables for which INSERT3 has not been called yet. Initially, the leaves of N are the n variables sorted in ascending order of *min*. Each non terminal node o contains a pointer to its two children $o.left$ and $o.right$.

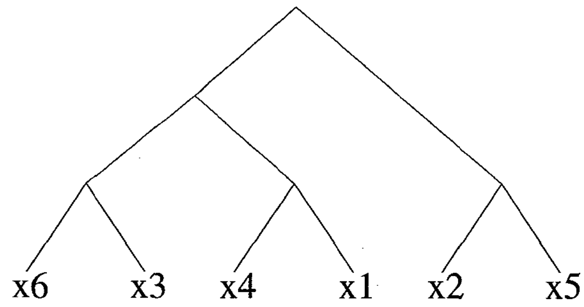

Figure 1: N tree in initial state.

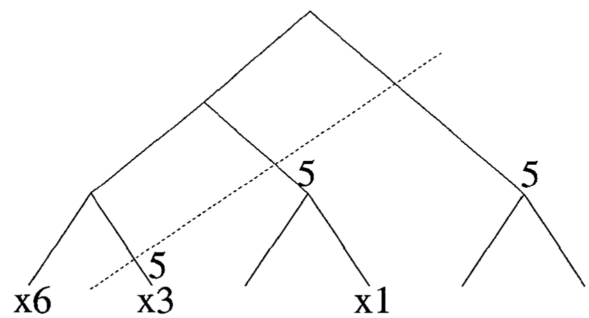

Figure 2: N tree after 3 iterations.

Figure1 represents N in its initial state for the timetabling example.

Within the function INSERT3 , a call to INCRMIN3(a,b) stores the information that the variables appearing in N that have a min greater than or equal to a must have a min greater than b. In order to do so, each node o contains a number *o.newmin*, initially set to 0, that represents the new min computed by the algorithm for all the leaves below o. The function INCRMIN3(a,b) sets *newmin*(o) to b for all the nodes o such that: all the leaves below o contain variables whose min is at least a, and the father of o has not been updated. In other words, INCRMIN3 only updates a "frontier" in N: all the leaves appearing on the right and below that frontier should be updated.

In our time tabling example, the first call to INCRMIN3 happens in the third call to INSERT3 , after the processing of x_4, x_2 and x_5. Then INCRMIN3(*2,4*) is called on N. The effect of this call is depicted on figure 2. It will store 5 as a new min for the node containing x_3 and the node ancestor of x_1. The dotted line indicates the frontier drawn by this call. All the nodes below and on the right of that frontier will have a min greater than or equal to 5.

```
% N, P are a binary trees
IncrMin3(a,b)
  o ← root of N
  while 1 do
    if o.min ≥ a then
      o.newmin ← b + 1
      return
    else
      o.right.newmin ← b + 1
      if o is a leaf then return
      o ← o.left

Extract(y)
  o ← root of N
  nmin ← 0
  while o is not a leaf do
    if o.newMin > nmin then nmin ← o.newMin
    o ← Select(o, y)
  if nMin > 0 then post y ≥ nmin

Select(o, y)
  if o.right.rank > y.rank then o ← o.left
  else o ← o.right
```

Algorithm 4: $O(nlog(n))$ filtering (cont'd)

The function EXTRACT(y) removes a variable y from N. It also traverses all the nodes on the path from the root of N to the leaf containing y, and collects the maximum *newmin* on that path. It then posts the constraint $y \geq newmin$. It also uses the function SELECT(o,y), which returns the child node of o that contains y in one of its leaves. In order to implement SELECT, each node o also contains the rank *o.rank* of its leftmost leaf. Then a simple test on the relative ranks of y and the right son of o is sufficient to decide whether y is below the left or the right son of o.

In our time tabling example, the fourth call of EXTRACT extracts x_3 from N, as depicted in figure 2. From the root to the node containing x_3, the maximum of the newmin is 5, hence $x_3 \geq 5$ is posted.

Note that each call to EXTRACT or INCRMIN3 traverses one path from the root of N to a leaf of N. It is a property of balanced binary trees that the length of such a path is at most $log(n)$. Thus, the functions EXTRACT(a)nd INCRMIN3 run in $O(log(n))$ time.

Using similar ideas, the function INSERT3 can also be implemented in $O(log(n))$. In order to introduce its implementation, we need to define for each variable y the following number:

$y.u = min(y) + \#(\{z | z \notin N, min(z) \geq min(y)]\})$

INSERT3 will update these numbers in a lazy way, using another balanced binary tree P. Initially, the leaves of P are empty. At the end of the algorithm, the leaves are all the variables, sorted in ascending order of min. The function INSERT3(y) inserts y in the *y.rank* leaf of P. Each node o in P contains a number *o.u* which is the

```
Insert3(y)
  o ← root(P)
  InsertAux(y,o,0,0)
  if o.u ≥ max(y) then IncrMin3(o.min,o.u)
InsertAux(y,o,delta,count)
  if o is a leaf then
    o.u ← count + min(y)
    o.min ← min(y)
    o.count ← 1
    o.delta ← 0
  else
    delta ← delta + o.delta
    o.min ← o.min + delta
    o.count ← o.count + 1
    o.delta ← 0
    if o.min ≤ min(y) then o.u ← o.u + 1
    if o.right.rank > y.rank then
      o.right.delta ← o.right.delta + delta
      count ← count + o.right.count
      InsertAux(y, o.left, delta, count)
    else
      o.left.delta ← o.left.delta + delta + 1
      InsertAux(y, o.right, delta, count)
    Update(o)
Update(o)
  uleft ← o.left.u + o.left.delta
  uright ← o.right.u + o.right.delta
  if uleft ≥ uright then
    o.u ← uleft
    o.min ← o.left.min
  else
    o.u ← uright
    o.min ← o.rigth.min
```

Algorithm 5: $O(nlog(n))$ filtering (cont'd)

maximum of $u(y)$ for the variables y appearing below o. o also contains the number *o.min* which is the min of the variable y such that $y.u = o.u$. If there exists several such variables, the one with the smallest min is selected. Intuitively, *o.y* is the best candidate for forming a Hall interval. When INSERT3(*x[i]*) is called, then *y.u* should be increased by 1 for all the variables appearing in P such that such that $min(y) < min(x[i])$. As the leaves are sorted by ascending order of *min*, it is sufficient to store this increment in the number *o.delta*, for nodes in that frontier. The only thing that remains to be computed is $x[i].u$. This number is equal to the number of variables y appearing in P such that $min(y) >= min(x[i])$. In order to do this, each node in P contains the number *o.count* of variables below it. Then, when inserting $x[i]$, it is sufficient to sum up the numbers *o.count* appearing on the right of the path to $x[i]$. Since a call to INSERT3 basically traverses one path from the root of P to one of its leaves, its complexity is $O(log(n))$.

6. Experimental results

In order to evaluate the actual usefulness of our work, we implemented algorithm 3 (let's call it algorithm A), and we compared it with 3 other algorithms on a set of various examples. The first algorithm (let's call it algorithm B) we considered is the basic one presented in the introduction: when a variable is assigned to a value, then this value is removed from the domain of all the other variables appearing in the constraint. Let's call Leconte's algorithm C and Regin's algorithm D. We chose to implement algorithm A using the Ilog Solver C++ library, as this library already provides an efficient implementation of algorithms B, C, and D. As all algorithms are implemented in the same library and run on the same computer, only their relative performance is of interest here. We report experiments running on a sparc 20 workstation.

Before presenting the results, we must say that our first implementation was not competitive at all: the overhead of manipulating binary trees was such that we obtained speedups only for constraints involving more than 10000 variables. After some further analysis of the algorithm, we decided to implement the following optimizations. The most effective optimization is to run algorithm B before, and to ignore the fixed variables. The second optimization is to treat all the variables having the same bounds in a single call to INSERT3 . All in all, these two optimizations improved the algorithm enough to be competitive even on small problems. Similar optimizations are used in algorithm C.

Experiment 1 is to apply each algorithm to the theoretical example used in the proof of proposition 2. This example was designed to show the worst case behavior of each algorithm. The results are summarized in the table below. First column gives the size of the problem, whereas each subsequent column gives the running time of the algorithms on that problem. We see that algorithm A has an almost linear running time on this example. Algorithm C and D clearly are quadratic (running time is multiplied by 4 each time the problem size doubles).

size	A	B	C	D
100	0.01	0.01	0.03	.07
200	0.01	0.07	0.08	.3
400	0.02	0.27	0.33	1.2
800	0.08	1.05	1.3	4.7
1600	0.15	4.2	5.3	18.8
3200	0.33	16.9	21.4	75.6
6400	0.77	125.9	122.7	>200
12800	1.7	>200	>200	
25600	3.5			
52800	7.7			
102400	15.8			

In the rest of the examples, we apply a MAC like algorithm, i.e. a backtracking algorithm where local consistency is applied at each node of the search tree. We ran a first set of examples where our algorithm A is used for filtering the alldiff constraint appearing on the example. Then we ran the same set of experiments using algorithm B for the alldiff constraints, and so on. For each experiment we indicate the running time and also the number of backtracks needed to solve the problem.

The next experiment we consider it is to find all solutions of the nqueen problem, represented as follows, using 3n variables and 3 alldiff constraints.

$\forall i, 1 \leq i \leq n,\ x_i \in [1,n],\ y_i \in [-n,n],\ z_i \in [1,2n],\ y_i = x_i - i,\ z_i = x_i + i,\ alldiff(x),\ alldiff(y), alldiff(z)$

In the table below, the row beginning with time8 gives the running time for finding all the solutions of the 8 queens problem. The row beginning with bt8 indicates the number of backtracks for the same set of experiences. The rows time9 and bt9 give the same information for the 9 queens problem, and so on. We can see that algorithms A, C, and D are almost useless here because although their improved pruning reduces the number of backtracks the total running time is longer than when using algorithm B. We can also notice that our algorithm is quite as fast as algorithm C and produces the same amount of pruning.

queens	A	B	C	D
time8	.23	.17	.21	.26
time9	.86	.64	.84	.98
tim10	3.4	2.6	3.3	3.8
time11	15.1	11.3	14.8	16.9
time12	73.9	54.7	71.8	82
bt8	260	289	260	239
bt9	947	1111	949	854
bt10	4294	5072	4295	3841
bt11	18757	22124	18763	16368
bt12	87225	103956	87263	74936

Experiment 3 is to find one solution to the nqueen problem, using a first fail principle for variable ordering. This one involves as many variables as we want. We notice that as in experiment 2, algorithm B is the fastest. As the problem size grows, we also see that the smallest complexity of algorithm A pays off.

1stqueen	A	B	C	D
time50	.13	.09	.12	.28
time100	.49	.31	.54	1.54
time200	1.9	1.2	2.5	9.7
time400	8.1	4.7	14.0	68.1
time800	38.3	19.2	91.6	>100

Experiment 4 is to solve the Golomb [3] problem to optimality. A Golomb problem of size n, involves $n(n+1)/2$ variables appearing in an alldiff

constraint, plus n^2 arithmetic constraints: $\forall i, 1 \leq i \leq n,\ x_i \in [1, 2^{n-1} - 1]$, $\forall i, j, i < j,\ y_{ij} = x_j - x_i$, $alldiff(y_{12}, ..., y_{n-1n})$, $minimize(x_n - x_1)$
In this example we see that in terms of pruning power, algorithm A is very similar to algorithm C, and lies between algorithms B and D. The speed difference between A and C, although small, increases with the size of the problem, due to the *nlog(n)* vs. n^2 complexity of the algorithms.

Golomb	A	B	C	D
time8	1.02	2.38	1.10	1.44
time9	7.50	22.4	8.10	10.8
tim10	59.8	210.9	64.8	88.6
tim11	1288		1430	
bt8	697	2735	697	697
bt9	3740	19445	3740	3740
bt10	23464	140746	23464	23464
bt11	374888		374888	

Experiment 5 uses a sport league scheduling problem described in McAloon [4]. Problem of size n involves scheduling $2n$ teams, and has $n(n+1)/2$ variables, $2n$ alldiff constraints involving $n$variables each, and one alldiff constraint involving $n(n+1)/2$ variables. The use of algorithm B did not lead to a solution within 10 minutes for n greater than 8. As in the previous experiment, the relative speed of algorithm A vs algorithm C increases with the size of the problem.

ttabling	A	B	C	D
t8	0.14	1.0	0.13	0.27
t10	1.5		1.5	2.4
t12	.47		.48	2.8
t14	18.5		22.3	39.4
t16	8.0		9.5	30.7
bt8	32	767	32	32
bt10	417		417	417
bt12	41		41	41
bt14	3514		3514	3508
bt16	1112		1112	1110

Several conclusions can be drawn from the experiments above. Algorithm A and algorithm C have almost the same pruning power, although Leconte's algorithm computes a stronger consistency than bound consistency. On small problems the overhead of using binary trees is not very important. On larger problems, the logarithmic behavior pays off. The comparison with algorithm D is quite unfair, because this algorithm is not suited for ordered domains. We also noticed that on easy problems, such as the n queens problem, the use of sophisticated algorithms does not pay at all.

7. Conclusion

We presented a global filtering algorithm for a very useful n-ary constraint, the alldiff constraint. A mathematical analysis of the constraint led us to introduce the notion of Hall intervals. We then derived a simple $O(n^3)$ algorithm for bound consistency that looped over all Hall intervals. We then proved that some Hall intervals are not useful for computing bound consistency, thereby reducing the running time complexity of the algorithm to $O(n^2)$. We then showed that using balanced binary trees, the same algorithm could be implemented in $O(nlog(n))$ running time. The resulting algorithm has been implemented and tested on various examples including theoretical ones and complex real examples. Results show that the actual running time of the algorithm is competitive compared to some of the best known algorithms.

As noticed in [3], the alldiff constraint can be seen as a special case of the one resource scheduling problem. A potential line of research is to investigate whether our algorithm can be extended to that case.

Acknowledgements This paper benefited from numerous discussions with Jean-Charles Regin, Michel Leconte, and all the ILOG Solver development team. The anonymous referee's remarks (especially Ω) greatly improved the accuracy of the paper.

References

1. Noelle Bleuzen Guernalec and Alain Colmerauer.
Narrowing a 2n-Bloc of sorting in O(nlog(n))
In *proceedings of CP'97, pages 2–16*, Linz, Austria, 1997.

2. P. Hall.
On representatives of subsets.
Journal of the London Mathematical Society 10:26-30, 1935.

3. M. Leconte
A Bounds-Based Reduction Scheme for Difference Constraints
in *proc. of Constraints96*, Key West, Florida, 19 May 1996.

4. Ken McAloon, Carol Tretkoff and Gerhard Wetzel
Sports league scheduling.
In *proceedings of the third Ilog Solver User's conference*, Paris, 1997.
Also available at http://www.ilog.com/ html/ products/ optimization/ customer_papers.htm

5. J-C. Régin.
A filtering algorithm for constraints of difference in CSPs.
In *proceedings of AAAI-94*, pages 362–367, Seattle, Washington, 1994.