A fast and simple algorithm for bounds consistency of the alldifferent constraint

Alejandro López-Ortiz, Claude-Guy Quimper,
John Tromp, Peter van Beek
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

Abstract

In constraint programming one models a problem by stating constraints on acceptable solutions. The constraint model is then usually solved by interleaving backtracking search and constraint propagation. Previous studies have demonstrated that designing special purpose constraint propagators for commonly occurring constraints can significantly improve the efficiency of a constraint programming approach. In this paper we present a fast, simple algorithm for bounds consistency propagation of the alldifferent constraint. In the general case the algorithm has the same worst case behavior as the previous best algorithms but is much faster in practice. Using a variety of benchmark and random problems, we show that our algorithm outperforms existing bounds consistency algorithms and also outperforms—on problems with an easily identifiable property—state-of-the-art commercial implementations of propagators for stronger forms of local consistency.

1 Introduction

Many interesting problems can be modeled and solved using constraint programming. In this approach one models a problem by stating constraints on acceptable solutions, where a constraint is simply a relation among several unknowns or variables, each taking a value in a given domain. The constraint model is then usually solved by interleaving backtracking search and constraint propagation. In constraint propagation the constraints are used to reduce the domains of the variables by ensuring that the values in their domains are locally consistent with the constraints.

Previous studies have demonstrated that designing special purpose constraint propagators for commonly occurring constraints can significantly improve the efficiency of a constraint programming approach (e.g., [8, 12]). In this paper we study constraint propagators for the alldifferent constraint. An alldifferent constraint over a set of variables states that the variables must be pairwise different. The alldifferent constraint is widely used in practice and because of its importance is offered as a builtin constraint in most, if not all, major commercial and research-based constraint systems.

Several constraint propagation algorithms for the alldifferent constraint have been developed, ranging from weaker to stronger forms of local consistency (see [15] for an excellent survey and see the next section for the definitions of the different forms of local consistency). Régin [8] gives an $O(n^{2.5})$ algorithm for domain consistency of the alldifferent constraint, where n is the number of variables in the constraint, that is based on relating alldifferent constraints to matchings. Leconte [4] gives an $O(n^2)$ algorithm for range consistency, a weaker form of local consistency than domain consistency, that is based on identifying Hall intervals. Puget [7], building upon the work of Leconte [4], gives an $O(n \log n)$ algorithm for bounds consistency, which is in turn a weaker form of local consistency than range consistency. Mehlhorn and Thiel [6], building upon the work of Régin [8], give an algorithm for bounds consistency that is $O(n)$ plus the time needed to sort the bounds of the domains, and thus has the same worst-case behavior as Puget's algorithm in the general case.

In this paper we present a fast and simple algorithm for bounds consistency propagation of the alldifferent constraint. In the general case the algorithm has the same worst case behavior as the previous best algorithms but is much faster in practice. Using a variety of benchmark and random problems, we show that our algorithm outperforms existing bounds consistency algorithms and also outperforms—on problems with an easily identifiable property—state-of-the-art commercial implementations of propagators for stronger forms of local consistency.

2 Background

We first define the constraint satisfaction problem and the relevant definitions of local consistency and then define the task studied in this paper.

A *constraint satisfaction problem* (CSP) consists of a set of n *variables*, $\{x_1, \ldots, x_n\}$; a finite domain $dom(x_i)$ of possible *values* for each variable x_i and a collection of m *constraints*, $\{C_1, \ldots, C_m\}$. Each constraint C_i is a constraint over some set of variables, denoted by $vars(C)$, that specifies the allowed combinations of values for the variables in $vars(C)$. Given a constraint C, we use the notation $t \in C$ to denote a tuple t—an assignment of a value to each of the variables in $vars(C)$—that satisfies the constraint C. We use the notation $t[x]$ to denote the value assigned to variable x by the tuple t. A *solution* to a CSP is an assignment of a value to each variable that satisfies all of the constraints.

We assume in this paper that the domains are totally ordered. The minimum and maximum values in the domain $dom(x)$ of a variable x are denoted by $\min(dom(x))$ and $\max(dom(x))$, and the interval notation $[a, b]$ is used as a shorthand for the set of values $\{a, a+1, \ldots, b\}$.

CSPs are usually solved by interleaving backtracking search and constraint propagation. During the backtracking search when a variable is assigned a value, constraint propagation ensures that the values in the domains of the unassigned variables are "locally consistent" with the constraints.

Definition 1 (Support) Given a constraint C, a value $a \in dom(x)$ for a variable $x \in vars(C)$ is said to have:

(i) a *domain support* in C if there exists a $t \in C$ such that $a = t[x]$ and $t[y] \in dom(y)$, for every $y \in vars(C)$;

(ii) an *interval support* in C if there exists a $t \in C$ such that $a = t[x]$ and $t[y] \in [min(dom(y)), max(dom(y)]$, for every $y \in vars(C)$.

Definition 2 (Local Consistency) A constraint C is said to be:

(i) *bounds consistent* if for each $x \in vars(C)$, each of the values $\min(dom(x))$ and $\max(dom(x))$ has an interval support in C.

(ii) *range consistent* if for each $x \in vars(C)$, each value $a \in dom(x)$ has an interval support in C.

(iii) *domain consistent* if for each $x \in vars(C)$, each value $a \in dom(x)$ has a domain support in C.

A CSP can be made locally consistent by repeatedly removing unsupported values from the domains of its variables.

Example 1 (Puget [7]) Consider the CSP with six variables $x_1, \ldots, x_6$; with the following domains, $x_1 \in [3,4]$, $x_2 \in [2,4]$, $x_3 \in [3,4]$, $x_4 \in [2,5]$, $x_5 \in [3,6]$, and $x_6 \in [1,6]$; and a single constraint alldifferent$(x_1, \ldots, x_6)$. Enforcing bounds consistency on the constraint reduces the domains of the variables as follows: $x_1 \in [3,4]$, $x_2 \in [2]$, $x_3 \in [3,4]$, $x_4 \in [5]$, $x_5 \in [6]$, and $x_6 \in [1]$.

Given an alldifferent constraint over n variables, where each of the variables $x_i, 1 \leq i \leq n$, has an interval domain $dom(x_i) = [\min_i, \max_i]$, our task is to make the constraint bounds consistent. More precisely, we must update the domain of each variable x_i to $dom'(x_i) = [\min'_i, \max'_i]$, the minimum and maximum value with interval support, or report that no solution exists.

3 Our result

In this section we present our algorithm for bounds consistency propagation of the alldifferent constraint and analyze its worst-case complexity.

3.1 Finding Hall intervals

Following Leconte [4] and Puget [7], we analyze the task in terms of *Hall intervals*. An interval I is a Hall interval if its size equals the number of variables whose domain is contained in I. Clearly, any solution must use all the values in I for those variables, making these values unavailable for any other variable. Puget shows that an algorithm for updating lower bounds can also be used to update upper bounds: simply invert each domain $[\min_i, \max_i]$ to $[-\max_i, -\min_i]$ before running the algorithm and invert the results back afterwards. The lower bound for variable x_i gets updated where $\min'_i \geq b+1$, whenever a Hall interval $[a, b]$ with $a \leq \min_i \leq b < \max_i$ is found. Note that this condition implies that any Hall interval $[a', b]$ with $a' < a$ causes the same update. Thus, for the purpose of updating lower bounds, it suffices to restrict attention to *left-maximal* Hall intervals: those $[a, b]$ for which a is minimal.

Puget's algorithm starts with sorting the variables in non-decreasing order of $\max_i$. Let us assume for convenience that $\max_i \leq \max_j$, for $i < j$. Puget's algorithm then processes each of the variables in turn, maintaining a set of counters which count how many of the variables processed so far have a minimum bound of at least k. More precisely, after processing x_i, the counter c_k^i denotes the cardinality of the set $\{j \leq i : \min_j \geq k\}$. The algorithm stores the counters in a balanced binary tree, allowing updates in $O(\log n)$ time per variable.

Conceptually, our algorithm is similar to Puget's. The difference is in the maintenance of the counters. The key observation is that not all counters are relevant.

Define $v_k^i = \max_i + 1 - k - c_k^i$ as the *capacity* of k after processing x_i. Intuitively this corresponds to as of yet unused values in the current interval.

If $v_k^i = 0$ and $k \leq \max_i$ then we have c_k^i equal to the cardinality of interval $[k, \max_i]$, which is thus recognized as a Hall interval. Capacities increase when $\max_{i+1} > \max_i$ and decrease when $\min_{i+1} \geq k$. More precisely, from the definition of capacity we have,

$$v_k^{i+1} - v_k^i = \max_{i+1} - \max_i - (c_k^{i+1} - c_k^i).$$

Now observe that c_k^{i+1} is the cardinality of the set $\{j \leq i+1 : \min_j \geq k\} = \{j \leq i : \min_j \geq k\} \cup \{j = i+1 : \min_j \geq k\}) = c_k^i + \delta$, where $\delta = 1$ if $k \leq \min_{i+1}$ and 0 otherwise. Hence,

$$v_k^{i+1} = v_k^i + \max_{i+1} - \max_i - \delta. \tag{1}$$

This shows the following.

Lemma 1 (Domination) *Let $k < k'$. If $v_k^i \leq v_{k'}^i$ then $v_k^{i'} \leq v_{k'}^{i'}$ for any $i' > i$.*

Capacity $v_{k'}^i$ is *dominated* by v_k^i in the sense that the former cannot reach 0 before the latter, and if both reach 0, then the Hall interval starting at k' is not left-maximal. If k is not equal to any $\min_i$, then it is always dominated by the next greater $\min_i$, hence we need only remember capacities for which k

equals some $\min_i$. The *critical set* C is the set of such indices of undominated capacities. This set starts out as $C^0 = \bigcup\{\min_i\}$ and becomes smaller over time as variables are processed and capacities become dominated. We denote by C^i the critical set after processing x_i. The next lemma shows that we can effectively test when each particular capacity v_k^i becomes zero or negative.

Lemma 2 (ZeroTest) *Let $k, l \in C^i$ be successive critical indices, such that $k \leq min_i < l$ and let $d = v_k^i - v_l^i$ be their difference in capacity. Suppose that for all $\min_i < m \leq \max_i$, capacity $v_m^i \geq 0$. Then when v_k^i is positive we have $\max_i + 1 - l + d \geq 2$ and when v_k^i is zero or negative we have $v_k^i = \max_i + 1 - l + d$.*

Proof: Since v_l is not dominated by v_k, Lemma 1 implies $d > 0$. By definition, $v_l^i = \max_i + 1 - l - c_l^i$. Now either $c_l^i = 0$ and then $v_k^i = v_l^i + d = \max_i + 1 - l + d$, or $c_l^i > 0$, which implies that for some $j \leq i$, $l \leq \min_j$. In that case, $l \leq \max_i$, which implies both (by assumption of the lemma) $v_l^i \geq 0$, giving $v_k^i = v_l^i + d \geq 1$, and $\max_i + 1 - l + d \geq 1 + d \geq 2$.

Our algorithm maintains the critical set C as a linked list in which each critical index points back to the preceding one, which, as shown above, has a larger capacity. The update rule (Equation 1) increases all capacities $v_k^{i-1}, k > \min_i$, by a certain amount, namely, $\max_{i+1} - \max_i - \delta$, and all capacities $v_k^{i-1}, k \leq \min_i$, by 1 less than that amount. This means that differences between adjacent critical capacities remain constant, except in one place: between capacities v_k and v_l of the ZeroTest Lemma, where the difference is reduced by 1. Therefore, testing for a zero or negative capacity need only be done at this v_k. Our linked list data structure is designed to perform this operation efficiently. All dominated indices form forests pointing toward the next critical index. A dummy index at the end, which never becomes dominated (3 larger than the largest max suffices), ensures that every dominated index has a critical one to point to. When a difference in capacity, say between indices $k_1 < k_2$, is reduced from 1 to 0, k_2 becomes dominated. It must then point to the next critical index, say k_3, which instead of pointing to k_2 must now point to k_1.

Example 2 Consider once again the CSP introduced in Example 1. The variables are already ordered by non-decreasing max. After placing a final dummy index at 9, we get initial capacities (relative to max = 4) of

$$v_1^0 = 4, v_2^0 = 3, v_3^0 = 2, v_9^0 = -4.$$

We represent this with the data structure

$$1 \xleftarrow{1} 2 \xleftarrow{1} 3 \xleftarrow{6} 9,$$

which only tracks the differences between adjacent critical capacities. Initially, these are exactly the differences between the indices themselves. Processing $x_1 \in [3, 4]$ reduces the difference between v_3 and v_9 from 6 to 5:

$$1 \xleftarrow{1} 2 \xleftarrow{1} 3 \xleftarrow{5} 9.$$

Processing $x_2 \in [2,4]$ reduces the difference between v_2 and v_3 from 1 to 0, causing v_3 to become dominated by v_2:

$$1 \xleftarrow{1} 2 \xleftarrow{5} 9.$$

Processing $x_3 \in [3,4]$ reduces the difference between v_2 and v_9 from 5 to 4:

$$1 \xleftarrow{1} 2 \xleftarrow{4} 9.$$

The zero capacity $v_2 = \max{+}1{-}9{+}4 = 0$ signals a Hall interval $[2, \max] = [2,4]$. Processing $x_4 \in [2,5]$ increases max to 5 while reducing the difference between v_2 and v_9 from 4 to 3:

$$1 \xleftarrow{1} 2 \xleftarrow{3} 9.$$

The zero capacity $v_2 = \max{+}1{-}9{+}3 = 0$ signals a Hall interval $[2, \max] = [2,5]$. Processing $x_5 \in [3,6]$ increases max to 6 while reducing the difference between v_2 and v_9 from 3 to 2:

$$1 \xleftarrow{1} 2 \xleftarrow{2} 9.$$

The zero capacity $v_2 = \max{+}1{-}9{+}2 = 0$ signals a Hall interval $[2, \max] = [2,6]$. Finally, processing $x_6 \in [1,6]$ reduces the difference between v_1 and v_2 from 1 to 0, causing v_2 to become dominated by v_1:

$$1 \xleftarrow{2} 9.$$

The zero capacity $v_1 = \max{+}1{-}9{+}2 = 0$ signals a Hall interval $[1, \max] = [1,6]$.

3.2 Updating bounds

Finding Hall intervals is only part of the solution. We also need to efficiently update the bounds. For this we will use another linked list structure, in which indices inside a Hall interval will point to the location representing its upper end, while those outside of any Hall interval will point left toward the next such index. We store the list of bounds in an array named `bounds`. The linked list is implemented in a secondary array `t` using indices to the `bounds` as pointers. The differences between critical capacities appearing above the arrows in Example 2 are stored in an array `d`. The algorithm shown in Figure 1 solves one half of the problem: updating all lower bounds.

Variable `niv` holds the number of intervals, while `nb` holds the number of unique bounds. The algorithm uses the following arrays:

- `maxsorted[0..niv-1]` is the array of intervals sorted by max.
- `bounds[0..nb+1]` is a sorted array of all min's and max's.
- `t[0..nb+1]` holds the critical capacity pointers; that is, `t[i]` points to the predecessor of i in the bounds list.

```
for (i = 1; i <= nb+1; i++) {
  t[i] = h[i] = i-1
  d[i] = bounds[i] - bounds[i-1]
}
for (i = 0; i < niv; i++) {
  (x, y) = (maxsorted[i]->minrank, maxsorted[i]->maxrank)
  z = pathmax(t, x+1)
  j = t[z]
  if (--d[z] == 0) {
    t[z] = z+1
    z = pathmax(t, t[z])
    t[z] = j
  }
  pathset(t, x+1, z, z)
  if (d[z] < bounds[z] - bounds[y])
    return Failure
  if (h[x] > x) {
    w = pathmax(h, h[x])
    maxsorted[i]->min = bounds[w]
    pathset(h, x, w, w)
  }
  if (d[z] == bounds[z] - bounds[y]) {
    pathset(h, h[y], j-1, y)
    h[y] = j-1;
  }
}
return Success
```

Figure 1: Algorithm for updating lower bounds.

- `d[0..nb+1]` holds the differences between critical capacities; that is `d[i]` is the difference of capacities between `bounds[i]` and its predecessor element in the list `bounds[t[i]]`.
- `minrank` and `maxrank` give the index in array `bounds` of the min and (max+1) of an interval.
- `h[0..nb+1]` holds the Hall interval pointers; that is, if `h[i]`$< i$ then the half-open interval [`bounds[h[i]]`,`bounds[i]`) forms a Hall interval. If the value is greater than i then it is a pointer to the Hall interval it belongs to. This Hall interval is represented by a tree, with the root containing the value of its right end.

The algorithm uses two functions for retrieving/updating pointer information:

- `pathmax(a, x)` follows the chain `x`,`a[x]`,`a[a[x]]`,..., until it stops increasing, returning the maximum found.

- `pathset(a, x, y, z)` sets each of the entries `a[x]`,`a[a[x]]`,...,`a[w]` to `z`, where `w` is such that `a[w]` equals `y`.

The first for-loop initializes arrays `t`, `h` and `d`. The next one passes over each interval in turn, calling `pathmax(t, x+1)` to get the first critical index beyond the interval minimum (`bounds[z]`). The corresponding difference is decremented and if zero, then the next critical index has been found. Then `pathset(t, x+1, z, z)` performs path compression on the chain of pointers followed. The entry `bounds[y]` holds the current max$+1$, so the next test recognizes a negative capacity, indicating that the constraint is unsatisfiable. The test `h[x] > x` recognizes indices inside a Hall interval which end at `bounds[pathmax(h, h[x]-1)]`, and the minimum bound is updated accordingly, followed by a path compression. The next test recognizes a possible zero capacity at `bounds[j]`. A new Hall interval $[\texttt{bounds[j]}, \texttt{bounds[y]} - 1]$ is recorded with `pathset(h, h[y], j-1, y)` and `h[y] = j-1`.

Example 3 Consider once again the CSP from Examples 1 & 2. Table 1 shows a trace of the algorithm for updating lower bounds (Figure 1) when applied to the CSP. For each iteration, two graphs are shown. In the first one, the nodes are the elements of the vector `bounds`. The arrows illustrate the content of the vector `t` and the numbers over them are given by the vector `d`. The nodes of the second graph are also the values found in vector `bounds` but the arrows are given by the vector `h` that keeps track of the Hall intervals.

3.3 Time complexity

The running time of the algorithm is dominated by the various calls to `pathmax` and `pathset`. Since each chain followed in a `pathmax` call is also followed in a subsequent `pathset` call, we can restrict our analysis to the time spent in the latter. Consider the right-running chains in array `t`. The following Lemma shows that all but a logarithmic number of indices see a rise in the value of a certain potential function as a result of path compression.

Lemma 3 *For any increasing sequence $0 < i_1 < i_2 < \cdots < i_k < n$ there are at most $\log n$ elements i_j such that $\lfloor \log(i_k - i_j) \rfloor = \lfloor \log(i_{j+1} - i_j) \rfloor$.*

Proof: Note that the *potential* function $\lfloor \log(i_{j+1} - i_j) \rfloor$ can take on at most $\log n$ values. If $i_j < i_{j'}$ share the same potential, then $\lfloor \log(i_k - i_j) \rfloor \geq \lfloor \log(i_{j'+1} - i_{j'} + i_{j+1} - i_j) \rfloor \geq \lfloor \log(i_{j+1} - i_j) \rfloor + 1$. Hence all but the last index with this potential will see a rise.

Since the potential function can never exceed $\log n$ this implies that a linear number of path compressions can take only $O(n \log n)$ steps. The situation with array `h` is similar. It follows that the entire algorithm runs in time $O(n \log n)$.

The theoretical performance of the algorithm can be improved further by observing that the union operations are always performed over sets whose bounds

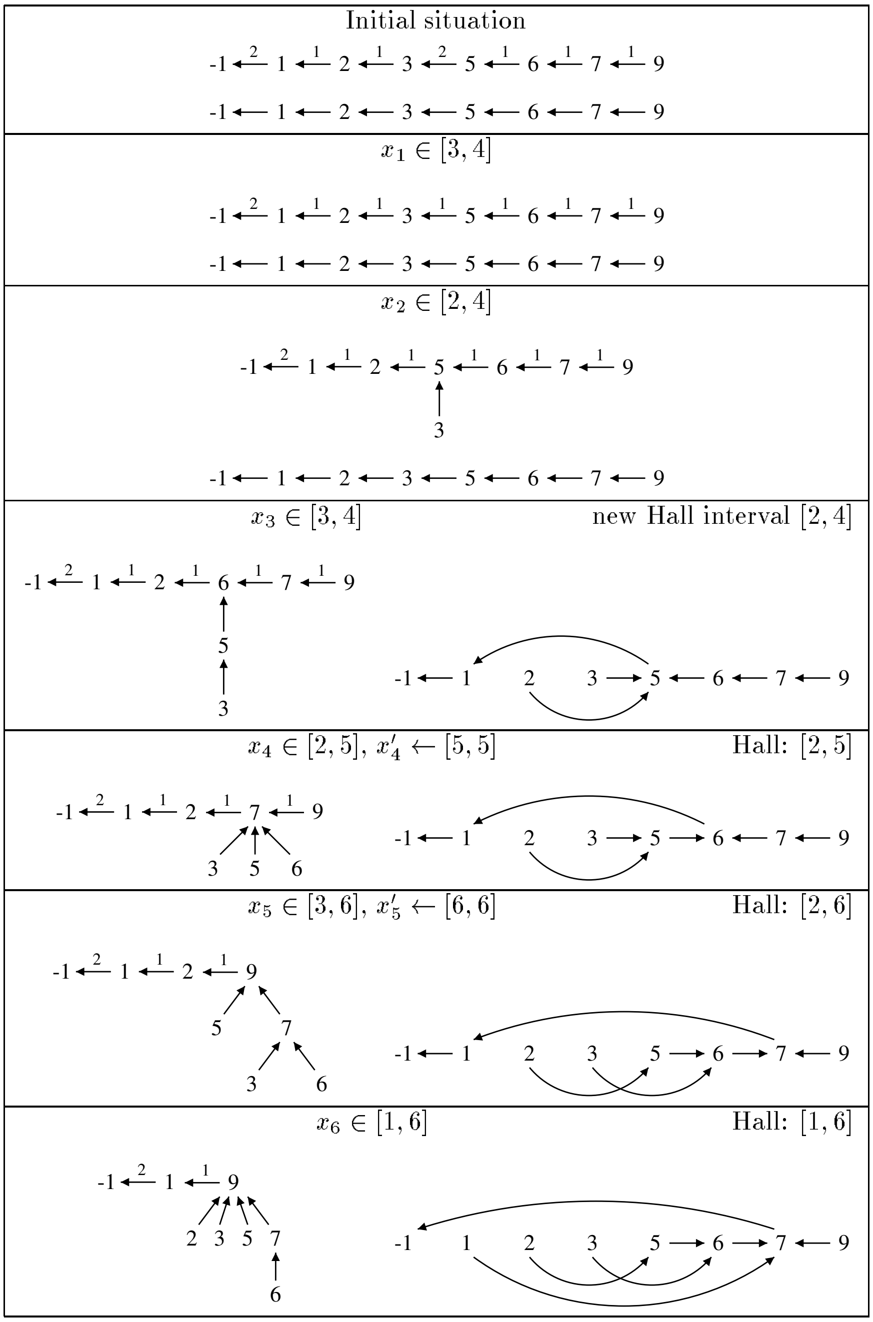

Table 1: Trace of the example: Each row represents an iteration where a variable is processed. The first graph illustrates the state for the vectors `bounds`, `t` and `d` while the second graph shows the state of the vectors `bounds` and `h`.

appear consecutively in a left to right ordering. This is known as the *interval union-find* problem. Gabow and Tarjan [1] gave a linear time solution in a RAM computer provided that the keys fit in a single word of memory. This is a reasonable assumption in current architectures with 32 or 64 bit words.

Using this technique we obtain a linear time algorithm which matches the theoretical performance of Melhorn and Thiel's solution. We implemented this algorithm on the Intel x386 architecture using the BSF (bit scan forward) and BSR (bit scan reverse) instructions using direct assembly code calls from a C++ program. However, in practice, the $O(n \log n)$ solution outperformed both Melhorn and Thiel's algorithM and the algorithm using the interval union find data structure.

4 Experimental results

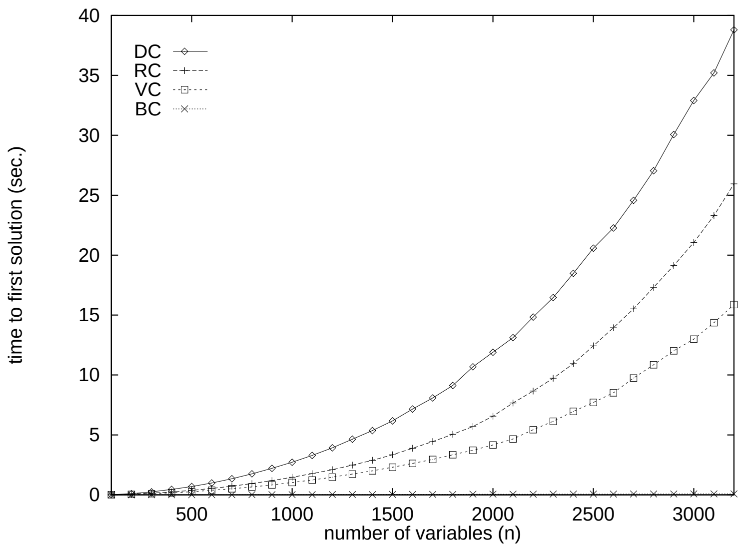

Figure 2: Time (sec.) to first solution for Pathological problems.

We implemented our new bounds consistency algorithm (denoted hereafter as BC) and Mehlhorn and Thiel's [6] bounds consistency algorithm (denoted MT) using the ILOG Solver C++ library, Version 4.2 [3][1]. The ILOG Solver library already provides implementations of Leconte's [4] range consistency algorithm (denoted RC), Régin's [8] domain consistency algorithm (denoted DC), and an algorithm that simply removes the value of an instantiated variable from the domains of the remaining variables (denoted as VC, for value consistency).

[1]The code discussed in this section is available on request from vanbeek@uwaterloo.ca

To compare against Puget's [7] bounds consistency algorithm, we use the runtime results reported by Puget [7] for RC and our own runtime results for RC as calibration points. We believe this is valid as Puget also uses a similar vintage of ILOG Solver and when we compared, we were careful to use the same constraint models and variable orderings.

We compared the algorithms experimentally on various benchmark and random problems. All the experiments were run on a 300 MHz Pentium II with 228 MB of main memory. Each reported runtime is the average of 10 runs except for random problems where more runs were performed.

We first consider a problem introduced by Puget ([7]; denoted here as Pathological problems) that were "designed to show the worst case behavior of each algorithm". A Pathological problem consists of a single alldifferent constraint over $2n+1$ variables with $dom(x_i) = [i-n, 0]$, $0 \leq i \leq n$, and $dom(x_i) = [0, i-n]$, $n+1 \leq i \leq 2n$. The problems were solved using the lexicographic variable ordering. On these problems, our BC propagator offers a clear performance improvement over propagators for stronger forms of local consistency (see Figure 2). Comparing against the best previous bounds consistency algorithms, our BC propagator is approximately 2 times faster than MT and, using RC as our calibration point to compare against the experimental results reported by Puget [7], approximately 5 times faster than Puget's algorithm.

m	VC	RC	DC	MT	BC
8	0.9	0.5	0.6	0.6	0.3
9	9.0	3.8	4.6	4.4	2.3
10	87.3	31.7	39.5	36.5	18.9
11	1773.6	688.1	871.2	841.4	437.6

Table 2: Time (sec.) to optimal solution for Golomb ruler problems.

We next consider the Golomb ruler problem (see [2], Problem 6). Briefly, a Golomb ruler is a set of m integers $0 = a_1 < a_2 \cdots < a_m$ such that the $m(m-1)/2$ differences $a_j - a_i$, $1 \leq i < j \leq m$ are distinct. Such a ruler is said to contain m marks and to be of length a_m. Following Smith et al. [11] we modeled the problem using auxiliary variables (the "ternary and all-different model" in [11]) and we used the lexicographic variable ordering. This appears to be the same model as Puget [7] uses in his experiments as the number of fails for each problem and each propagator are the same. On these problems, our BC propagator is approximately 1.6 times faster than the next fastest propagator used in our experiments (see Table 2) and, again using RC as our calibration point, approximately 1.5 times faster than Puget's bounds consistency algorithm.

We next consider instruction scheduling problems for single-issue processors with arbitrary latencies. Instruction scheduling is one of the most important steps for improving the performance of object code produced by a compiler. Briefly, in the model for these problems there are n variables, one for each instruction to be scheduled, latency constraints of the form $x_i \leq x_j + d$ where d is

n	VC	RC	DC	MT	BC
69		0.02	0.04	0.04	0.02
70		0.02	0.05	0.04	0.02
111	0.08	0.07	0.12	0.11	0.07
211					
214	40.64	0.67	1.20	0.94	0.46
216		0.31	1.08	0.72	0.38
220		0.29	0.93	0.66	0.32
377		0.84	3.94	2.41	0.79
381	0.28	0.50	3.18	1.15	0.39
394		2.66	7.15	2.66	1.65
556					
690		1.91	26.88	3.95	1.61
691		14.47	40.50	6.30	3.14
856		7.72	17.09	10.86	5.48
1006		10.35	87.23	15.90	5.95

Table 3: Time (sec.) to optimal solution for instruction scheduling problems. A blank entry means the problem was not solved within a 10 minute time bound.

some small integer value, a single alldifferent constraint over all n variables, and redundant constraints called "distance constraints" (see [13] for more details; additional redundant constraints are identified there which we omit in this study in order to focus on the alldifferent constraint). In our experiments, we used fifteen representative hard problems that were taken from the SPEC95 floating point, SPEC2000 floating point and MediaBench [5] benchmarks. The minimum domain size variable ordering heuristic was used in the search (see Table 3). On these problem too, our BC propagator offers a clear performance improvement over the other propagators.

In order to systematically study the scaling behavior of the algorithms, we next consider random problems. The problems consisted of a single alldifferent constraint over n variables and each variable x_i had its initial domain set to $[a, b]$, where a and b, $a \leq b$, were chosen uniformly at random from $[1, n]$. The problems were solved using the lexicographic variable ordering. In these "pure" problems nearly all of the run-time is due to the alldifferent propagators, and one can clearly see the quadratic behavior of the RC and DC propagators and the nearly linear incremental behavior of the BC propagator (see Figure 3). On these problems, VC (not shown) could not solve even the smallest problems (n = 100) within a 10 minute time bound and MT (also not shown) was 2.5 – 3 times slower than our BC propagator.

Having demonstrated the practicality of our algorithm, we next study the limits of its applicability. Schulte and Stuckey [10] investigate cases where it can be proven a priori that maintaining bounds consistency during the backtracking

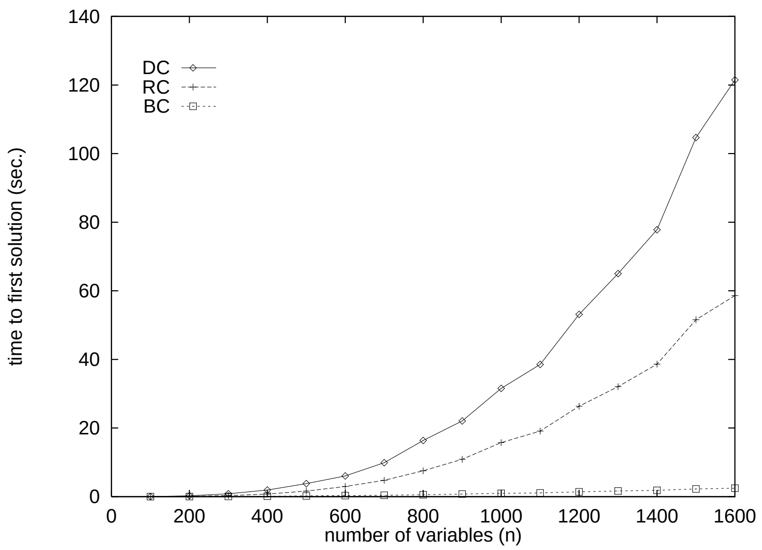

Figure 3: Time (sec.) to first solution or to detect inconsistency for random problems. Each data point is the average of 100 problems.

search, rather than a stronger form of local consistency such as range or domain consistency, does not increase the size of the search space. The Golomb ruler problem is one such example. In general, of course, this is not the case and using bounds consistency can exponentially increase the search space.

In order to systematically study the range of applicability of the algorithms, we next consider random problems with holes in the domains of the variables. The problems consisted of a single alldifferent constraint over n variables. The domain of each variable was set in two steps. First, the initial domain of the variable was set to $[a, b]$, where a and b, $a \leq b$, were chosen uniformly at random from $[1, n]$. Second, each of the values $a + 1, \ldots, b - 1$ is removed from the domain with some given probability p. The resulting problems were then solved using both the lexicographic and the minimum domain size variable ordering heuristics. These problems are trivial for domain consistency, but not so for bounds and range consistency. We recorded the percentage that were not solved by BC and RC within a fixed time bound (see Figure 4). If there are no holes in the domains of the variables, then bounds consistency is equivalent to range and domain consistency. As the number of holes increases, the performance of bounds and range consistency decreases and they become less appropriate choices. The range of applicability of BC can be extended somewhat following a suggestion by Puget [7] of combining bounds consistency with value consistency (denoted hereafter as BC+ and MT+; see Figure 4).

We next consider the well-known n-queens problem. We use the model and variable ordering heuristic given in [3]. Briefly, the model consists of $3n$ variables

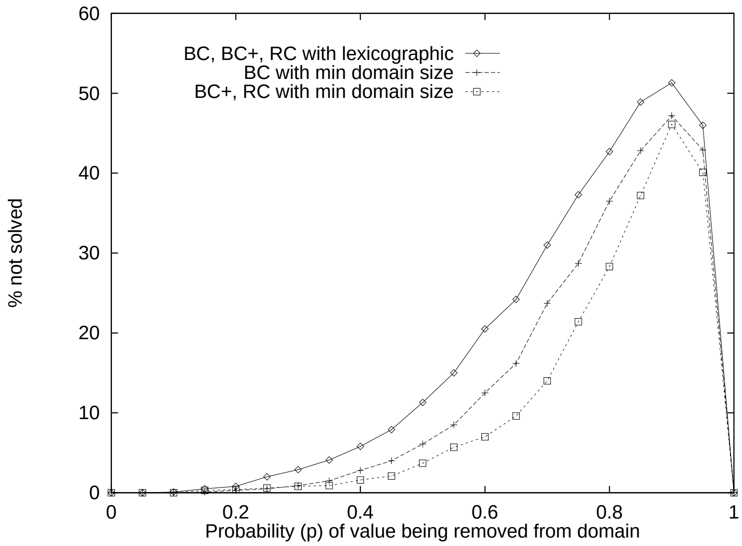

Figure 4: Percentage not solved within 5 seconds for problems with 100 variables. Each data point is the average of 100 problems. The cutoff (5 seconds) was chosen to be the value that was at least two orders of magnitude slower than domain consistency, the fastest propagator on these problems.

with three alldifferent constraints over n variables each and the variable ordering heuristic consists of choosing the variable with the minimum domain size and breaking ties by choosing the variable with the smallest minimum domain value (see Table 4). On these problems, reasoning about the holes in the domains is crucial and bounds consistency alone, both BC and MT, cannot solve them. BC+ and MT+ can solve them but are not competitive with VC alone.

We next consider the Quasigroup existence problems (see [2], Problem 3 and the code given there). Following Stergiou and Walsh [12] the problem is modeled using $2n$ alldifferent constraints over n variables each, where n is the order of the problem. Each axiom defines additional constraints. The problems were solved using the minimum domain size variable ordering heuristic (see Table 5). On these problems too, reasoning about the holes in the domains is crucial and bounds consistency alone, both BC and MT, cannot solve them. BC+ and MT+ can solve them but are not competitive with DC.

Interestingly, in our experiments, RC was never the propagator of choice. On problems where holes arise in the domains, DC was the best choice (except for on n-queens problems, where VC was considerably faster), and on problems where holes do not arise in the domains, BC was the clear best choice. Clearly, whether the domains have holes in them is a property that is easily identified and tracked during the search. Thus, the best choice of propagator could be automatically selected, rather than left to the constraint modeler to specify as

	n	VC	RC	DC	MT$^+$	BC$^+$
all	8	0.0	0.0	0.1	0.2	0.1
	9	0.2	0.2	0.3	0.6	0.3
	10	0.6	0.8	1.0	2.3	1.0
	11	2.7	3.5	4.4	10.4	4.6
	12	13.1	17.0	21.6	52.3	22.6
first	100	0.1	0.1	0.4	0.2	0.1
	200	0.3	0.5	2.4	1.0	0.5
	400	1.1	2.9	17.5	4.3	1.9
	800	5.7	20.3	160.0	22.0	10.1
	1600	32.5	164.4	1264.2	111.8	56.2

Table 4: Time (sec.) to find first or all solutions for n-queens problems.

is currently the case.

5 Conclusions

We presented an improved bounds consistency constraint propagation algorithm for the important alldifferent constraint. Using a variety of benchmark and random problems, we showed that our algorithm significantly outperforms the previous best bounds consistency algorithms for this constraint and can also significantly outperform propagators for stronger forms of local consistency.

6 Acknowledgements

We thank Kent Wilken for providing the instruction scheduling problems used in our experiments.

References

[1] H. N. Gabow and R. E. Tarjan. A Linear-Time Algorithm for a Special Case of Disjoint Set Union. *Journal of Computer and System Sciences,* 30(2): 209-221, 1985.

[2] I. P. Gent and T. Walsh. CSPlib: A benchmark library for constraints. In *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming*, pages 480–481, Alexandria, Virginia, 1999.

[3] ILOG S. A. ILOG Solver 4.2 user's manual, 1998.

[4] M. Leconte. A bounds-based reduction scheme for constraints of difference. In *Proceedings of the Constraint-96 International Workshop on Constraint-Based Reasoning*, pages 19–28, Key West, Florida, 1996.

axiom	order	VC	RC	DC	MT^+	BC^+
3	8	0.2	2.9	2.7	0.2	0.2
	9	773.8	630.5	557.0	667.3	641.6
4	8	9.0	7.5	6.9	8.1	7.7
	9	1.6	1.3	0.8	1.4	1.4
5	8	0.2	0.1	0.2	0.1	0.1
	9	1.1	0.9	0.9	0.9	0.9
	10	14.4	12.0	6.3	12.3	12.1
	11	67.1	21.1	9.7	21.5	21.2
	12	1077.2	708.9	231.6	721.9	712.9
6	8	0.0	0.0	0.0	0.0	0.0
	9	0.2	0.2	0.2	0.2	0.2
	10	2.0	1.9	1.9	2.1	2.0
	11	32.4	31.7	32.1	33.6	32.0
	12	842.1	815.8	820.2	864.6	829.5
	13	10.0	10.0	10.3	10.8	10.3
7	8	0.9	0.8	0.9	1.0	0.9
	9	2.1	2.1	2.3	2.4	2.2
	10	499.8	483.8	506.0	529.9	495.2

Table 5: Time (sec.) to first solution for Quasigroup problems. For each axiom, problems of order 8, ..., 13 were solved. The absence of an entry means the problem was not solved within a 1 hour time bound.

[5] C. Lee, M. Potkonjak, and W. Manginoe-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications. In *Proceedings of International Symposium on Microarchitecture*, pages 330–335, 1997.

[6] K. Mehlhorn and S. Thiel. Faster algorithms for bound-consistency of the sortedness and alldifferent constraint. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming*, pages 306–319, Singapore, 2000.

[7] J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 359–366, Madison, Wisconsin, 1998.

[8] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 362–367, Seattle, 1994.

[9] J.-C. Régin. Modeling and solving sports league scheduling with constraint programming. In *INFORMS*, Cincinnati, Ohio, 1999.

[10] C. Schulte and P. J. Stuckey. When do bounds and domain propagation lead to the same search space. In *Proceedings of the Third International*

Conference on Principles and Practice of Declarative Programming, pages 115–126, Firenze, Italy, 2001.

[11] B. M. Smith, K. Stergiou, and T. Walsh. Using auxiliary variables and implied constraints to model non-binary problems. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 182–187, Austin, Texas, 2000.

[12] K. Stergiou and T. Walsh. The difference all-difference makes. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 414–419, Stockholm, 1999.

[13] P. van Beek and K. Wilken. Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, pages 625–639, Paphos, Cyprus, 2001.

[14] P. Van Hentenryck, L. Michel, L. Perron, and J.-C. Régin. Constraint programming in OPL. In *Proceedings of the First International Conference on Principles and Practice of Declarative Programming*, pages 98–116, Paris, 1999.

[15] W. J. van Hoeve. The alldifferent constraint: A survey, 2001. Submitted manuscript. Available from http://www.cwi.nl/~wjvh/papers/alldiff.pdf.