An Efficient Bounds Consistency Algorithm for the Global Cardinality Constraint

Claude-Guy Quimper, Peter van Beek, Alejandro López-Ortiz, Alexander Golynski, and Sayyed Bashir Sadjad

School of Computer Science
University of Waterloo
Waterloo, Canada

Abstract. Previous studies have demonstrated that designing special purpose constraint propagators can significantly improve the efficiency of a constraint programming approach. In this paper we present an efficient algorithm for bounds consistency propagation of the generalized cardinality constraint (*gcc*). Using a variety of benchmark and random problems, we show that our bounds consistency algorithm is competitive with and can dramatically outperform existing state-of-the-art commercial implementations of constraint propagators for the *gcc*. We also present a new algorithm for domain consistency propagation of the *gcc* which improves on the worst-case performance of the best previous algorithm for problems that occur often in applications.

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 problem is then usually solved by interleaving a backtracking search with a series of constraint propagation phases. In the constraint propagation phase, the constraints are used to prune 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., [7, 11]). In this paper we study constraint propagators for the global cardinality constraint (*gcc*). A *gcc* over a set of variables and values states that the number of variables instantiating to a value must be between a given upper and lower bound, where the bounds can be different for each value. This type of constraint commonly occurs in rostering, timetabling, sequencing, and scheduling applications (e.g., [1, 4, 9, 13]).

Two constraint propagation techniques for the *gcc* have been developed. Régin [8] gives an $O(n^2d)$ algorithm for domain consistency of the *gcc* (where n is the number of variables and d is the number of values) that is based on relating the *gcc* to flow theory. As well, a *gcc* can be rewritten as a collection of “atleast” and “atmost” constraints, one for each value, and constraint propagation can be performed on the individual constraints [14]. However, on some problems the first technique suffers from its cubic run-time and the second technique suffers from its lack of pruning power. An alternative which has not yet been explored with the *gcc* is bounds consistency propagation, a weaker form of consistency than domain consistency. Bounds consistency propagation has already proven useful for the *alldifferent* constraint [5, 10], a specialization of the *gcc*.

In this paper we present an efficient algorithm for bounds consistency propagation of the *gcc*. Using a variety of benchmark and random problems, we show that our bounds consistency algorithm is competitive with and can dramatically outperform existing state-of-the-art commercial implementations of constraint propagators for the *gcc*. We also present a new algorithm for domain consistency propagation of the *gcc* which improves on the worst-case performance of Régin's algorithm for problems that occur often in applications.

2 Background

A *constraint satisfaction problem* (CSP) consists of a set of n *variables*, $X = \{x_1, \ldots, x_n\}$; a set of d *values*, $D = \{v_1, \ldots, v_d\}$, where each variable $x_i \in X$ has an associated finite domain $dom(x_i) \subseteq D$ of possible values; 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 $vars(C_i)$. Given a constraint C, the notation $t \in C$ denotes a tuple t—an assignment of a value to each of the variables in $vars(C)$—that satisfies the constraint C. The notation $t[x]$ denotes 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 a backtracking search with constraint propagation. The constraint propagation phase ensures that the values in the domains of the unassigned variables are “locally consistent” with the constraints.

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 $y \in vars(C)$.

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) *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.

A *global cardinality constraint (*gcc*)* is a constraint which consists of a set of variables $X = \{x_1, \dots, x_n\}$, a set of values $D = \{v_1, \dots, v_d\}$, and for each $v \in D$ a pair $[l_v, u_v]$. A *gcc* is satisfied iff the number of times that a value $v \in D$ is assigned to the variables in X is at least l_v and at most u_v.

Example 1. Consider the CSP with six variables $x_1, \dots, x_6$ with domains, $x_1 \in [2, 2]$, $x_2 \in [1, 2]$, $x_3 \in [2, 3]$, $x_4 \in [2, 3]$, $x_5 \in [1, 4]$, and $x_6 \in [3, 4]$ and a single global cardinality constraint $gcc(x_1, \dots, x_6)$ with bounds on the occurrences of values, $l_1, l_2, l_3 = 1, l_4 = 2$ and $u_v = 3$, for all $v \in \{1, 2, 3, 4\}$. Enforcing bounds consistency on the constraint reduces the domains of the variables as follows: $x_1 \in [2, 2]$, $x_2 \in [1, 1]$, $x_3 \in [2, 3]$, $x_4 \in [2, 3]$, $x_5 \in [4, 4]$, and $x_6 \in [4, 4]$.

3 Local consistency of the *gcc*

A *gcc* can be decomposed into two constraints: A *lower bound constraint* (*lbc*) which ensures that all values $v \in D$ are assigned to at least l_v variables, and an *upper bound constraint* (*ubc*) which ensures that all values $v \in D$ are assigned to at most u_v variables. We will show how to make both constraints locally (bounds or domain) consistent and prove that this is sufficient to make a *gcc* locally consistent.

3.1 The Upper Bound Constraint (*ubc*)

The *ubc* is a generalization of the well studied *alldifferent* constraint (in the *alldifferent* constraint $u_v = 1$, for each value v). Some previous algorithms for bounds consistency of the *alldifferent* constraint have been based on the concept of Hall intervals [3, 5, 6]. A Hall interval is an interval $H \subseteq D$ such that there are $|H|$ variables whose domains are contained in H. The definition of a Hall interval can be generalized to sets by using the notion of maximal capacity. Let $C(S)$, $S \subseteq D$, be the number of variables whose domains are contained in S. The maximal capacity $\lceil S \rceil$ of a set S is the maximum number of variables that can be assigned to the values in S; i.e., $\lceil S \rceil = \sum_{v \in S} u_v$.

Hall set A Hall set is a set $H \subseteq D$ such that there are $\lceil H \rceil$ variables whose domains are contained in H; i.e., H is a Hall set if $C(H) = \lceil H \rceil$.

The values in a Hall set are fully consumed by the variables that form the Hall set and unavailable for all other variables. Clearly, a *ubc* is unsatisfiable if there is a set S such that $C(S) > \lceil S \rceil$. We show that the absence of such a set is a sufficient and necessary condition for a *ubc* to be satisfiable.

Lemma 1. *A* ubc *is satisfiable if and only if for any set* $S \subseteq D$, $C(S) \leq \lceil S \rceil$.

Proof. We reduce a *ubc* to an *alldifferent* constraint. We first duplicate u_v times each value v in the domain of a variable, using different labels to represent the same value. For example, the domain $\{1, 2\}$ with $u_1 = 3$ and $u_2 = 2$ is represented by $\{1a, 1b, 1c, 2a, 2b\}$. Clearly, this *alldifferent* constraint is satisfiable iff the *ubc* is satisfiable. In a *ubc*, the maximal capacity of a set S is given by $\lceil S \rceil$; in an *alldifferent* constraint, it is given by the cardinality $|S|$ of the set. Hall [3] proved that an *alldifferent* constraint is satisfiable if and only if for any set S, $C(S) \leq |S|$. Thus, the result holds also for a *ubc*. $\square$

3.2 The Lower Bound Constraint (*lbc*)

Next we define some concepts that will be useful for constructing a propagator for the *lbc*. Let $I(S)$ be the number of variables whose domains intersect the set S. The minimal capacity $\lfloor S \rfloor$ of a set S is the minimum number of variables that must be assigned to the values in S; i.e., $\lfloor S \rfloor = \sum_{v \in S} l_v$.

Failure set A failure set is a set $F \subseteq D$ such that there are fewer variables whose domains intersect F than its minimal capacity; i.e., F is a failure set if $I(F) < \lfloor F \rfloor$.

Unstable set An unstable set is a set $U \subseteq D$ such that there are the same number of variables whose domains intersect U as its minimal capacity; i.e., U is an unstable set if $I(U) = \lfloor U \rfloor$.

Stable set A stable set is a set $S \subseteq D$ such that there are more variables whose domains are contained in S than its minimal capacity, and S does not intersect any failure or unstable sets; i.e., S is a stable set if $C(S) > \lfloor S \rfloor$, $S \cap U = \emptyset$ and $S \cap F = \emptyset$ for all unstable sets U and failure sets F.

These three sets are the main tools to understand how to make an *lbc* locally consistent. Failure sets determine if an *lbc* is satisfiable, unstable sets indicate where the domains have to be pruned, and stable sets indicate which domains do not have to be pruned because all of their values have supports.

Lemma 2. *An* lbc *is satisfiable if and only if it does not have a failure set.*

Proof. To satisfy an *lbc*, we must associate at least l_v different variables to each value $v \in D$ such that every variable is assigned a single value from its domain. For each value $v \in D$, we construct l_v identical sets T_v^i for $i = 1, \ldots, l_v$ that contain the indices of the variables that have v in their domain; i.e., $T_v^i = \{j \mid x_j \in X \wedge v \in dom(x_j)\}$. Let $\mathcal{T}$ be the set of all sets T_v^i. To satisfy the *lbc*, we must select one variable index from each set T_v^i such that all selected indices are different. The variables that are not selected can be instantiated to any arbitrary value in their domain. This problem is known as the complete set of distinct representatives problem and has been studied by Hall [3]. His main result states that for any family of sets, a complete set of distinct representatives exists if and only if the union of any k sets contains at least k elements. Formally the problem is solvable if and only if $|\bigcup_{t \in T} t| \geq |T|$ holds for any $T \subseteq \mathcal{T}$. Applying this theorem here, we have that an *lbc* is satisfiable if and only if for any set $S \subseteq D$ we have $I(S) \geq \lfloor S \rfloor$. Hence, the absence of a failure set is a necessary and sufficient condition for an *lbc* to be satisfiable. $\square$

Lemma 3 shows that a value in a domain that intersects an unstable set has an interval/domain support only if the value also is in the unstable set.

Lemma 3. *A variable whose domain intersects an unstable set cannot be instantiated to a value outside of this set.*

Proof. Let U be an unstable set and x a variable whose domain intersects U. If x is instantiated to a value that does not belong to U then U becomes a failure set and the *lbc* is no longer satisfiable by Lemma 2. □

Lemma 4. *A variable whose domain is contained in a stable set can be instantiated to any value in its domain.*

Proof. By definition, a stable set S does not intersect any unstable or failure set. Thus, for any subset s of S, $I(s) > \lfloor s \rfloor$. If a variable whose domain is contained in S is assigned a value, the function $I(s)$ will decrease by at most one and therefore s will either stay a stable set or become an unstable set. In both cases, no failure set is created and the *lbc* is still satisfiable. □

A satisfiable *lbc* has several interesting properties: (i) the union of two unstable sets gives an unstable set, (ii) the union of two stable sets gives a stable set, and (iii) since stable and unstable sets are disjoint, there exists a stable set S and an unstable set U that forms a bipartition of D. The bipartition property implies that there are two types of variables: those whose domains are fully contained in a stable set and those whose domains intersect an unstable set.

3.3 An Iterative Algorithm for Local Consistency of the *gcc*

Suppose we have an algorithm $\mathcal{A}$ that makes a *ubc* locally consistent and suppose that we have an algorithm $\mathcal{B}$ that makes an *lbc* locally consistent. To make a *gcc* locally consistent we can decompose it, run $\mathcal{A}$ to prune the domains of the variables, and then run $\mathcal{B}$ to further prune the domains. Since the domains can potentially be pruned each time either algorithm is run, we alternatively run each algorithm until no more modifications occur. In principle, we might need to repeat this process a large number of times. Surprisingly, we prove that only one iteration is sufficient.

The outline of the proof is as follows. We first prove that if a *ubc* is satisfiable after running $\mathcal{A}$, the *ubc* is still satisfiable after running $\mathcal{B}$. We then prove that the *ubc* is still locally consistent after running $\mathcal{B}$.

Theorem 1. *If $\mathcal{B}$ is run after $\mathcal{A}$, $\mathcal{B}$ never creates a set s such that there are more variables whose domains are contained in s than its maximal capacity $\lceil s \rceil$.*

Proof. Suppose that algorithms $\mathcal{A}$ and $\mathcal{B}$ do not return a failure. Then there are no failure sets and there is an unstable set U and a stable set S that form a bipartition of D. Algorithm $\mathcal{B}$ does not modify the domains of the variables that belong to a stable set. Therefore we know that for all $s \subseteq S$ we have $C(s) \leq \lceil s \rceil$ since the *ubc* is satisfiable according to $\mathcal{A}$.

We will show that for any set $E \subseteq U \cup S$ we have $C(E) \leq \lceil E \rceil$ and therefore the *ubc* is still satisfiable after running $\mathcal{B}$. Assume, by way of contradiction, there is a set E that exceeds its capacity; i.e., $C(E) > \lceil E \rceil$. We divide this set into two subsets: let $L = U \cap E$ be the unstable values in E and $F = S \cap E$ be the stable values in E. We also define $R = U - E$ as the unstable values that do not belong to E. We know that $\lceil F \rceil \geq C(F)$ since F is a subset of a stable set and we showed that the property holds for any such a set. We also know that R is not a failure set and U is an unstable set. Therefore we have $I(R) \geq \lfloor R \rfloor$ and $\lfloor L \rfloor + \lfloor R \rfloor = I(L \cup R)$.

$$\begin{aligned}
\lceil F \rceil + \lfloor L \rfloor + \lfloor R \rfloor &\leq \lceil F \rceil + \lceil L \rceil + \lfloor R \rfloor \\
\lceil F \rceil + I(L \cup R) &< C(E) + \lfloor R \rfloor \\
\lceil F \rceil + I(L \cup R) &< |\{x \in X \mid x \subseteq E \wedge x \nsubseteq F\}| + C(F) + \lfloor R \rfloor \\
\lceil F \rceil + I(L \cup R) &< |\{x \in X \mid x \cap L \neq \emptyset \wedge x \cap R = \emptyset\}| + C(F) + \lfloor R \rfloor \\
\lceil F \rceil + I(R) &< C(F) + \lfloor R \rfloor \\
\lceil F \rceil &< C(F)
\end{aligned}$$

The last inequality is incompatible with the hypothesis hence the contradiction hypothesis cannot be true. Notice that the proof holds for both bounds and domain consistency. □

Theorem 2. *If $\mathcal{B}$ is run after $\mathcal{A}$, the* ubc *is still locally consistent after $\mathcal{B}$ is run.*

Proof. Suppose that $\mathcal{A}$ and $\mathcal{B}$ make the constraints locally consistent and neither returns a failure. To prove that the *ubc* is still locally consistent, we have to show that all variables are still consistent with all Hall sets. By a variable being consistent with a Hall set H we mean the following: for bounds consistency, the domain of the variable must have either both or neither bounds in H; and for domain consistency, the domain of the variable must be either fully included in or completely disjoint from H.

Since $\mathcal{B}$ did not return a failure, there is an unstable set U and a stable set S that form a bipartition of D. Let $H \subseteq D$ be a Hall set. We divide this Hall set into two subsets: $F = H \cap S$ contains the values of H that belong to a stable set and $L = H \cap U$ contains the values of H that belong to an unstable set. We also define $R = U - L$ as the unstable values that do not belong to H. Using these three sets, we will prove that all variables are consistent with H.

The unstable set U can be expressed as the union of L and R and therefore we have $\lfloor L \rfloor + \lfloor R \rfloor = I(L \cup R)$. Similarly, H is the union of F and L and implies $\lceil F \rceil + \lceil L \rceil = C(H) = |\{x \in X \mid x \subseteq H \wedge x \nsubseteq F\}| + C(F)$. Therefore we have

$$\begin{aligned}
\lceil F \rceil + \lfloor L \rfloor + \lfloor R \rfloor &\leq \lceil F \rceil + \lceil L \rceil + \lfloor R \rfloor \\
\lceil F \rceil + I(L \cup R) &\leq |\{x \in X \mid x \subseteq H \wedge x \nsubseteq F\}| + C(F) + \lfloor R \rfloor \\
\lceil F \rceil + I(L \cup R) &\leq |\{x \in X \mid x \cap L \neq \emptyset \wedge x \cap R = \emptyset\}| + C(F) + \lfloor R \rfloor \\
\lceil F \rceil + I(R) &\leq C(F) + \lfloor R \rfloor
\end{aligned}$$

By Theorem 1 we obtain $C(F) \leq \lceil F \rceil$ and since R is not a failure set, we have $I(R) \geq \lfloor R \rfloor$. Using these two inequalities, we find that R is an unstable set i.e. $I(R) = \lfloor R \rfloor$ and F is a Hall set i.e. $\lceil F \rceil = C(F)$. Using this observation, we now show that all variables that are contained in S are consistent with H. The Hall set F is a subset of S and since algorithm $\mathcal{B}$ does not modify any variables that are contained in S, algorithm $\mathcal{A}$ already identified F as a Hall set and already made all variables consistent with it. Since the variables contained in S were not modified by $\mathcal{B}$ they are still consistent with F. A variable that intersects an unstable set like U and R must have both bounds in this set. Since $U = L \cup R$, a variable that intersects U must have both bounds in either L or R and therefore be consistent with the Hall set H. Using the same scheme, one can show that the result also holds for domain consistency.

We have shown that all variables that are contained in S and those that intersect U are consistent with H. Thus all variables are consistent with any Hall set and the *ubc* is still locally consistent after running $\mathcal{B}$. □

Finally, we show that making the *ubc* and the *lbc* locally consistent is equivalent to making the *gcc* locally consistent.

Theorem 3. *A value $v \in dom(x)$ has a support in a* gcc *if and only if it has supports in the corresponding* lbc *and* ubc.

Proof. Clearly, if there is a tuple t that satisfies the *gcc* such that $t[x] = v$, this tuple also satisfies the *lbc* and the *ubc*. To prove the converse, we consider a value $v \in dom(x)$ that has a support in the *lbc* and a (possibly different) support in the *ubc*. We construct a tuple t such that $t[x] = v$ that satisfies the *gcc* and therefore prove that $v \in dom(x)$ also has a support in the *gcc*. We first instantiate the variable x to v. The *lbc* and *ubc* are still satisfiable since the value has a support in both constraints. We now show how to instantiate the other variables.

If there is an uninstantiated variable x whose domain does not intersect any unstable set and is not contained in any Hall set, then the domain of x is necessarily contained in a stable set. By Lemma 4 we can instantiate x to any value in its domain and keep the *lbc* satisfiable. We therefore choose a solution of the *ubc* and instantiate x to the same value as it is instantiated in the solution. This operation can create new unstable sets or new Hall sets but keeps both the *lbc* and the *ubc* satisfiable. For all variables that intersect an unstable set U, we choose a solution of the *lbc* and assign the variables to the same values as the solution. We perform the same operation for the variables whose domain is contained in a Hall set H using a solution of the *ubc*. There will be exactly l_v or u_v variables assigned to a value v depending if the value belongs to U or H, which in either case satisfies both the *lbc* and *ubc*. We repeat the above until all variables are instantiated. The constructed tuple t satisfies the *lbc* and the *ubc* simultaneously and therefore also satisfies the *gcc*. □

4 Bounds Consistency

We present algorithms for making a *ubc* and an *lbc* bounds consistent.

4.1 The Upper Bound Constraint (*ubc*)

Finding an algorithm that makes a *ubc* bounds consistent is relatively straightforward if we already know such an algorithm for the *alldifferent* constraint that uses the concept of Hall intervals. Such an algorithm first detects Hall intervals. If there is a variable whose domain is $[a, b]$ and there is a Hall interval $[c, d]$ such that $c \leq a \leq d < b$ holds, the algorithm will update the domain of the variable to $[d+1, b]$. The algorithm introduced in [5] detects Hall intervals by checking if there are $d - c + 1$ variables in an interval $[c, d]$. We can adapt this algorithm to a *ubc* without altering its complexity by finding a way to compute the maximal capacity of an interval in constant time. We use a partial sum data structure, implemented as an array A containing the partial sums of the maximal capacities $A[i] = \sum_{j=0}^{i} u_j$. The maximal capacity of an interval $I \subseteq D$ can be computed by subtracting two elements in A since we have $\lceil I \rceil = A[\max(I)] - A[\min(I) - 1]$. Initializing the array A takes $O(D)$ time to compute but this is done once and is reused for any future calls to the propagator.

4.2 The Lower Bound Constraint (*lbc*)

We now present an algorithm that makes an *lbc* bounds consistent (see Figure 1). The algorithm receives as input a set of intervals which represent the domains of the variables and returns as output the same intervals with their lower bounds updated. The upper bounds can be updated symmetrically by a similar algorithm. An *lbc* is bounds consistent once both bounds have been updated.

The initialization step of the algorithm iterates through each value $v \in D$, assigning l_v empty *buckets* corresponding to the minimal capacity to be filled for v and setting a *failure flag* which will indicate if v belongs to a failure set. The disjoint set data structure *PS* covers all values in D and contains potential stable sets. If the greatest element of a set $S \in PS$ is in a stable set, all elements in S belong to the same stable set. The variable *Stable* contains these stable sets.

To detect the stable sets, the algorithm processes each variable $x \in X$ in nondecreasing order by upper bound. Each time, it searches for the smallest value $v \in dom(x)$ that has an empty bucket and fills it in with a token. If $v > \min(dom(x))$ and v belongs to a stable set then the interval $I = [\min(dom(x)), v]$ also belongs to a stable set. The algorithm regroups all the values of I in its variable *PS* in case it discovers that v belongs to a stable set. If there are no empty buckets in $dom(x)$ then $\max(dom(x))$ belongs to a stable set and so do all the values that belong to the same set in *PS*.

To detect failure sets, the algorithm initially assumes that all values belong to a failure set. But when it discovers a stable or unstable set, it resets the failure flags for all values of this set. If a value still has a failure flag set after processing all the variables then the *lbc* is unsatisfiable. Non-failure sets are discovered when the last empty bucket of $\max(dom(x))$ is filled in. An interval $[a, \max(dom(x))]$ that only contains values without empty buckets cannot be a failure set and the algorithm unsets the failure flags for all these values.

```
Let PS be a disjoint set data structure over the elements in D;
Let Stable = ∅;
for v ∈ D do
    associate l_v empty buckets to the value v;
    mark v as a failure element;
D ← D ∪ {−∞, ∞};
associate ∞ buckets to the values −∞ and ∞;
for x_i ∈ X in nondecreasing order of max(dom(x_i)) do
    a ← min(dom(x_i)); b ← max(dom(x_i));
    z ← min({v ∈ D | v ≥ a, a has an empty bucket});
    if z > a then union (PS, a, a + 1, ..., min(b, z));
    if z > b then
        S ← findSet (PS, b);
        Stable ← Stable ∪ {S};
    else
        add a token in one of the empty buckets of z;
        z ← min({v ∈ D | v ≥ a, a has an empty bucket});
        NewMin[i] ← min({v ∈ D | v ≥ a, v has a failure flag});
        if z > b then
            j ← max({v ∈ D | v ≤ b, v has an empty bucket});
            reset the failure flag for all elements in (j, b];
if |{v ∈ D | v has a failure flag}| > 0 then return Failure;
for x_i ∈ X do
    a ← min(dom(x_i)); b ← max(dom(x_i));
    if |{S ∈ Stable | a ∈ S ∧ b ∈ S}| = 0 then
        dom(x_i) ← dom(x_i) − [a, NewMin[i]);
return Success;
```

Algorithm 1: Bounds consistency algorithm for the *lbc*

To actually update the domains, the algorithm stores in the array *NewMin* the potential new lower bound of each variable. This lower bound is equal to the smallest value in $dom(x)$ that has a failure flag. If no such value exists, $NewMin[x]$ is left undefined. After processing all the variables, the algorithm iterates over the variables a second time and updates the domain of any variable x that does not belong to a stable set, by setting its lower bound to $NewMin[x]$.

Example 2. We present a trace of the algorithm on the CSP introduced in Example 1. Initially, all buckets are empty and all values are marked with a failure flag. Figure 1 shows the data structures as the algorithm iterates through the variables. The circles represent the buckets, a letter f symbolizes a failure flag, and the state of the variables *PS* and *Stable* are also represented by the sets of values. Upon completion of the algorithm, the new domains of the variables are: $x_1 \in [2, 2]$, $x_2 \in [1, 2]$, $x_3 \in [2, 3]$, $x_4 \in [2, 3]$, $x_5 \in [4, 4]$, and $x_6 \in [4, 4]$.

To conclude the discussion of our bounds consistency algorithm for the *lbc*, we show how to obtain an efficient implementation of the algorithm. A naive

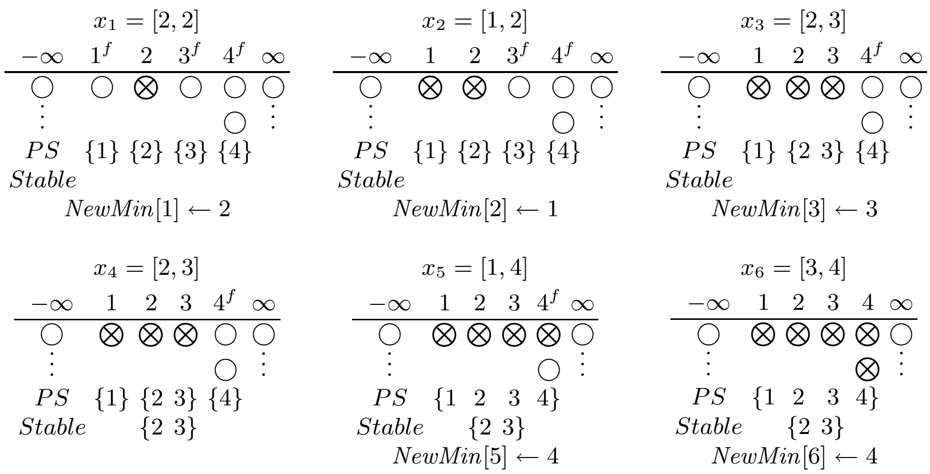

Fig. 1. Trace of Algorithm 1

implementation has time complexity $O(t + |X|\,|D|)$, where t is the complexity of sorting the intervals by upper bounds. The time for sorting can be smaller than $O(|X| \log |X|)$ if we use an incremental algorithm or a linear time sorting algorithm. We will show how to improve the complexity to $O(t + |X|)$.

We first observe that if $l_v = 0$ for many values of $v \in D$ then $|D|$ can be much greater than $|X|$. To obtain a complexity independent of $|D|$, we consider the variables as semi-open intervals where $x_i = [a_i, b_i)$ and define the set D' as the union of the lower bounds a_i and the open upper bounds b_i of each variable. Obviously, the new set D' contains a maximum of $2|X|$ elements. Let c and d be two consecutive values in D' and let $I = (c, d]$ be a semi-open interval. We modify the algorithm to assign $\lfloor I \rfloor$ buckets to the value d using a partial sum data structure (see Section 4.1). We then run the algorithm as before using the set D' instead of D. This modification improves the time complexity to $O(t + |X|^2)$.

The second improvement concerns the data structures. The buckets can be implemented with a disjoint set data structure and an array of integers that stores the number of empty buckets a value v has. If a value has no more free buckets after adding a token, the algorithm merges the value v with the next element in D'. Requesting n times the next value having a free bucket is a linear time operation using the interval union-find data structure [2]. Using the same data structure for the failure flags, the stable sets *Stable*, and the potential stable sets *PS* the algorithm takes $O(t + |X|)$ steps.

Although the interval union-find data structure gives the best theoretical time complexity, we found that it did not result in the fastest code in practice in spite of our best efforts to optimize the code. We use instead the tree data structure described in [5] to obtain an algorithm with $O(t + |X| \log |X|)$ time complexity.

5 Domain Consistency

In this section we present a propagator that makes a *gcc* domain consistent. We will use Régin's propagator [7, 15] for the *alldifferent* constraint as a black box that has complexity $O(d|X|^{\frac{3}{2}})$, where d is the size of the largest domain of a variable, to make the *lbc* and *ubc* domain consistent.

5.1 The Upper Bound Constraint (*ubc*)

The problem of making a *ubc* domain consistent can be reduced to the problem of making an *alldifferent* constraint domain consistent. Consider the domain $dom(x)$ of a variable x as a multiset where the multiplicity of a value $v \in dom(x)$ is u_v. One can represent a multiset as a normal set where different labels refer to the same value. We apply Régin's propagator with the new domains and then remove all duplicates from the domains. Since there are $|X|$ variables and the largest domain is bounded by $u|D|$ where $u = \max_{v \in D} u_v$, we obtain a time complexity of $O(u\,|D|\,|X|^{\frac{3}{2}})$.

5.2 The Lower Bound Constraint (*lbc*)

The problem of making an *lbc* domain consistent can also be reduced to the problem of making an *alldifferent* constraint domain consistent. We first duplicate the values as we did in Section 5.1 according to the minimal capacities. Let M be a $|X| \times |D|$ binary matrix such that M_{ij} equals 1 if the value j belongs to the domain of the variable x_i and equals 0 otherwise. The transposed matrix M^T defines the dual problem. In a dual problem, the dual values D' represent the primal variables and the dual variables X' represent the primal values.

Theorem 4. *Solving the* alldifferent *problem on the dual problem solves the lower bound problem on the primal problem.*

Proof. Since we have duplicated some values in the domains of the variables, the minimal capacity of a set S is now equal to the size of the set; i.e., $\lfloor S \rfloor = |S|$. Let U be an unstable set in the primal problem. In the dual problem, the values in U are represented by variables. There are $|U|$ dual variables whose domains are contained in a set of $|U|$ dual values, since U is an unstable set. Consequently, an unstable set in the primal corresponds to a Hall set in the dual. A propagator for the *alldifferent* problem removes from a domain the values contained in a Hall set only if the domain is not fully contained in the Hall set. If such a propagator is applied on the dual problem, it would remove from the domains that intersect an unstable set the values that are not fully contained in the unstable set. This operation is sufficient to make the problem domain consistent. The *alldifferent* propagator would also return a failure if the problem is unsolvable. A failure set in the primal problem corresponds to a set of values in the dual problem that contains more variables than values. Such a set makes the dual problem unsolvable and is detected by the *alldifferent* propagator. □

We use Régin's propagator to solve the dual problem and then remove the duplicates from the domains of the variables. Since in the dual problem there are at most $l|D|$ variables and the largest domain is bounded by $|X|$, the total time complexity is $O(l^{1.5}|X||D|^{1.5})$ where $l = max_{v \in D} l_v$.

5.3 Putting all Together

The complete algorithm makes the *ubc* domain consistent and then makes the *lbc* domain consistent. The total time complexity is $O(u|X|^{1.5}|D| + l^{1.5}|X||D|^{1.5})$.

That the complexity depends on the number of values in D can make the filter inefficient for some problems. We identify two classes of problems that occur often in applications and where our algorithm offers a better complexity than existing algorithms. Our analysis assumes that the maximal capacity u_v is bounded by a constant for all values v. The first class consists of problems where the minimal capacity l_v is non-null. Since each value must be instantiated by at least one variable, we necessarily have $|D| \leq |X|$ for a solvable problem. In this case the algorithm runs in time $O(|D||X|^{1.5})$. The second class of problems is the one where the minimal capacity l_v is null for all values v. In this case we only need to make *ubc* domain consistent which can be done in time $O(|D||X|^{1.5})$. For either class, the complexity of the algorithm improves the previous best *gcc* propagator for domain consistency which runs in $O(|D||X|^2)$ [8].

6 Experimental Results

We implemented our new bounds consistency algorithm for the generalized cardinality constraint (denoted hereafter as BC) using the ILOG Solver C++ library, Version 4.2 [4][1]. Following a suggestion by Puget [6] adapted to the *gcc*, the range of applicability of BC can be extended by combining bounds consistency with the removal of a value when the number of times it has been assigned reaches its upper bound (denoted BC+). The ILOG Solver library already provides implementations of Régin's [8] domain consistency algorithm (denoted DC), and an algorithm (denoted CC) that enforces a level of consistency that is equivalent to enforcing domain consistency on individual cardinality constraints, where there is one cardinality constraint for each value [4, 14].

We compared the algorithms experimentally on various benchmark and random problems. All of the experiments were run on a 2.40 GHz Pentium 4 with 1 GB of main memory. Each reported runtime is the average of 10 runs except for random problems where 100 runs were performed. Unless otherwise noted, the minimum domain size variable ordering heuristic was used in the search.

We first consider problems introduced by Puget ([6]; denoted here as Pathological) that were "designed to show the worst case behavior" of algorithms for the *alldifferent* constraint. Here we adapt the problem to the *gcc*. A Pathological problem consists of a single *gcc* over $2n + 1$ variables with $dom(x_i) = [i - n, 0]$,

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

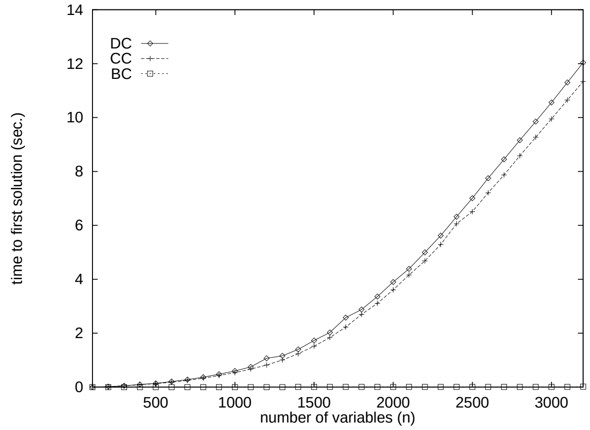

Fig. 2. Time (sec.) to first solution for Pathological problems.

$0 \leq i \leq n$, and $dom(x_i) = [0, i-n]$, $n+1 \leq i \leq 2n$ and each value must occur exactly once. The problems were solved using the lexicographic variable ordering. On these problems, our BC propagator offers a clear performance improvement over the other propagators (see Figure 2). Qualitatively similar results were obtained for a generalization of these problems where each value must occur exactly c times, where c is some small value.

We next consider instruction scheduling problems for multiple-issue pipelined processors. For these problems there are n variables, one for each instruction to be scheduled and latency constraints of the form $x_i \leq x_j + l$ where l is some small integer value, and one or more *gcc*'s over all n variables (see [12] for more details on the problem). In our experiments, we used ten hard problems that were taken from the SPEC95 floating point, SPEC2000 floating point, and MediaBench benchmarks (see Table 1). The issue width of a processor refers to how many instructions can be issued each clock cycle. In our experiments we used the representative cases of a processer with an issue width of two with two identical functional units (see Table 1(a)) and an issue width of four with two floating point units and two integer units (see Table 1(b)). Here, our BC propagator offers a clear performance improvement over the other propagators.

We next consider car sequencing problems (see [4]). For these problems there are n variables, n values, each configuration of five options is equally likely, and there are approximately $4n$ *gcc*'s. Here, our BC+ propagator achieves almost the same pruning power as DC and becomes faster than the other propagators as n grows (see Table 2). We also consider sport league scheduling problems (see [13] and references therein). For these problems there are n^2 variables, n values, and $n/2$ *gcc*'s. Here, our BC+ propagator is within 15% of the fastest propagator, DC, in terms of run-time and pruning power (see Table 3). The complexity or run-time of the CC and DC propagators depends on the number of domain values,

Table 1. Time (sec.) to optimal solution for instruction scheduling problems; (a) issue width = 2; (b) issue width = 2 + 2 = 4. A blank entry means the problem was not solved within a 10 minute time bound.

n	CC	DC	BC
69	0.01	0.12	0.00
70	0.00	0.07	0.00
111	0.03	0.75	0.01
211	0.51	9.24	0.07
214	0.60	9.29	0.09
216	2.67	124.07	0.31
220	5.09	285.91	0.52
690	1.34	493.15	1.67
856		471.16	3.84
1006			8.70

(a)

n	CC	DC	BC
69	0.00	0.07	0.00
70	0.01	0.07	0.00
111	0.03	0.44	0.01
211	0.56	7.16	0.11
214	0.61	7.85	0.13
216	2.78	89.61	0.48
220	2.90	98.15	0.57
690	2.17	307.20	2.81
856			
1006	307.00		14.44

(b)

whereas the BC/BC+ propagators do not. The car sequencing and sports league scheduling problems illustrate that the number of domain values does not have to be very large for this factor to lead to competitive run-times for our relatively unoptimized BC/BC+ propagators.

To systematically study the scaling behavior of the algorithm, we next consider random problems. The problems consisted of a single *gcc* over n variables and each variable had its initial domain set to $[a, b]$, where a and b, $a \leq b$, were chosen uniformly at random from $[1, d = n/2]$ (chosen so that a mixture of consistent and inconsistent problems would be generated). In these "pure" problems nearly all of the run-time is due to the *gcc* propagators, and one can clearly see the cubic behavior of the DC propagator and the nearly linear incremental behavior of the BC propagator (see Table 4). On these problems, CC (not shown) could not solve some of the smallest problems within a 10 minute time bound.

Table 2. (a) Time (sec.) to first solution or to detect inconsistency for car sequencing problems; (b) number of backtracks (fails).

n	CC	DC	BC	BC+
10	0.07	0.07	0.09	0.09
15	3.40	3.88	5.39	4.12
20	20.65	30.05	30.95	21.83
25	131.27	203.23	163.97	118.57

(a)

n	CC	DC	BC	BC+
10	437	321	460	429
15	13,849	9,609	19,958	13,565
20	55,657	52,581	105,436	55,580
25	255,690	250,042	520,519	255,653

(b)

Table 3. (a) Time (sec.) to first solution for sports league scheduling problems; (b) number of backtracks (fails). A blank entry means the problem was not solved within a 10 minute time bound.

n	CC	DC	BC	BC+
8	0.19	0.16	0.04	0.18
10	1.10	0.12	0.03	0.19
12	1.98	1.70	51.71	2.07
14	11.82	8.72		9.98

(a)

n	CC	DC	BC	BC+
8	1308	914	136	942
10	5767	428	54	689
12	6449	4399	149728	5356
14	33901	19584		22176

(b)

Table 4. Time (sec.) to first solution (or to detect inconsistency) for random problems where the bounds on number of occurrences of each value were (a) $[0, 2]$; (b) chosen uniformly at random from $\{[0, 1], [0, 2], [1, 1], [1, 2], [1, 3], [2, 2], [2, 3], [2, 4]\}$. A blank entry means some problems could not be solved within a 10 minute time bound.

n	DC	BC
100	0.02	0.01
200	0.23	0.02
400	2.55	0.08
800	26.14	0.33
1600	266.80	1.24

(a)

	DC			BC		
n	$d/2$	d	$2d$	$d/2$	d	$2d$
100	0.00	0.01	0.33	0.00	0.00	0.00
200	0.00	0.07	4.81	0.00	0.01	0.01
400	0.01	0.60	74.88	0.00	0.03	0.04
800	0.03	4.58		0.01	0.15	0.16
1600	0.20	34.78		0.02	0.70	0.62

(b)

7 Conclusions

We presented an efficient algorithm for bounds consistency propagation of the *gcc* and showed its usefulness on a set of benchmark and random problems. We also presented an algorithm for domain consistency propagation with an improved worst-case bound on problems that arise in practice.

Acknowledgments. The authors thank the participants of the constraint programming problem session at the University of Waterloo and Kent Wilken for providing the instruction scheduling problems used in our experiments.

References

1. Y. Caseau, P.-Y. Guillo, and E. Levenez. A deductive and object-oriented approach to a complex scheduling problem. In *Deductive and Object-Oriented Databases*, pages 67–80, 1993.
2. H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *STOC-1983*, pages 246–251.
3. P. Hall. On representatives of subsets. *J. of the London Mathematical Society*, pages 26–30, 1935.

4. ILOG S. A. ILOG Solver 4.2 user's manual, 1998.
5. A. López-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *IJCAI-2003*.
6. J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *AAAI-1998*, pages 359–366.
7. J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI-1994*, pages 362–367.
8. J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *AAAI-1996*, pages 209–215.
9. J.-C. Régin and J.-F. Puget. A filtering algorithm for global sequencing constraints. In *CP-1997*, pages 32–46.
10. C. Schulte and P. J. Stuckey. When do bounds and domain propagation lead to the same search space. In *PPDP-2001*, pages 115–126.
11. K. Stergiou and T. Walsh. The difference all-difference makes. In *IJCAI-1999*, pages 414–419.
12. P. van Beek and K. Wilken. Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In *CP-2001*, pages 625–639.
13. P. Van Hentenryck, L. Michel, L. Perron, and J.-C. Régin. Constraint programming in OPL. In *PPDP-1999*, pages 98–116.
14. P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58:113–159, 1992.
15. W. J. van Hoeve. The alldifferent constraint: A survey. Unpublished manuscript, 2001.