Constraints, 10, 115–135, 2005

An Efficient Bounds Consistency Algorithm for the Global Cardinality Constraint

CLAUDE-GUY QUIMPER cquimper@uwaterloo.ca
School of Computer Science, University of Waterloo, Waterloo, Canada

ALEXANDER GOLYNSKI agolynsk@uwaterloo.ca
School of Computer Science, University of Waterloo, Waterloo, Canada

ALEJANDRO LÓPEZ-ORTIZ alopez-o@uwaterloo.ca
School of Computer Science, University of Waterloo, Waterloo, Canada

PETER VAN BEEK vanbeek@waterloo.ca
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 on some problems our bounds consistency algorithm 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.

Keywords: global constraints, bounds consistency, domain 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 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., [12, 16]). 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., [2, 6, 14, 19]).

Two constraint propagation techniques for the *gcc* have been developed. Régin [13] 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 [20]. 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 was not 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 [9, 15], a specialization of the *gcc*.

Independently to our work, Katriel and Thiel [7] enforce bounds consistency by using the same technique as Régin but exploit the convexity of the graph to obtain an $O(t + n + d)$ algorithm where t is the time to sort n variable domains by lower and upper bounds. Their algorithm also enforces bounds consistency on the cardinality variables that restrict the number of variables that can be assigned to a same value. Quimper et al. [11] proved that enforcing domain consistency on the same variables is NP-Hard.

In this paper we present an efficient algorithm for bounds consistency propagation of the *gcc*. The algorithm runs in time $O(t + n)$, where t is the time to sort the bounds of the domains of the variables and n is the number of variables. Using a variety of benchmark and random problems, we show that on some problems our bounds consistency algorithm 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 integers. 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 $X = \{x_1, \ldots, x_n\}$, a set of values $D = \{v_1, \ldots, v_d\}$, 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 $x_1, \ldots, x_6$ 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 $gcc(x_1, \ldots, x_6)$ with bounds on the occurrences of values,

v	1	2	3	4
l_v	1	1	1	2
u_v	3	3	3	3

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 [4, 9, 10]. 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 iff $C(H) = \lceil H \rceil$.

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 [4] proved that an *alldifferent* constraint is satisfiable iff for any set S, $C(S) \leq |S|$. Thus, the result holds also for a *ubc*. ■

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.

In Example 1, the set $\{1, 4\}$ is an unstable set since its lower capacity is 3 and only 3 variable domains (namely x_2, x_5, and x_6) intersect it. The set $\{4\}$ is also an unstable set and $\{2, 3\}$ is a stable set. There are no failure sets in the example but removing variable x_2 would create the failure set $\{1, 4\}$.

Failure, unstable, and stable 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 [4]. 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 $|\cup_{t \in T} t| \geq |T|$ 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. ■

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.

Lemma 5 *If there are no failure sets, the union of two unstable sets gives an unstable set.*

Proof: Let U_1 and U_2 be two unstable sets. We have that,

$$I(U_1 \cup U_2) = I(U_1) + I(U_2) - I(U_1 \cap U_2) \tag{1}$$

$$= \lfloor U_1 \rfloor + \lfloor U_2 \rfloor - I(U_1 \cap U_2). \tag{2}$$

Since there are no failure sets we have $I(U_1 \cup U_2) \geq \lfloor U_1 \rfloor + \lfloor U_2 \rfloor - \lfloor U_1 \cap U_2 \rfloor$. We also have $I(U_1 \cap U_2) \geq \lfloor U_1 \cap U_2 \rfloor$. Substituting these two inequalities in Equation 2 gives $I(U_1 \cup U_2) = \lfloor U_1 \cup U_2 \rfloor$. ■

Lemma 6 *If there are no failure sets, there exists a bipartition* $\langle U, S \rangle$ *of D where U is an unstable set and S is a stable set.*

Proof: Let U be the union of all unstable sets. By Lemma 5, U is also an unstable set. Since there are no failure sets we have $I(D) \geq \lfloor D \rfloor$. Suppose that $I(D) = \lfloor D \rfloor$, then $U = D$ and $S = \emptyset$. Now suppose that $I(D) > \lfloor D \rfloor$. We have that,

$$\begin{aligned} C(D - U) &= |X| - I(U) \\ &= |X| - \lfloor U \rfloor \\ &> \lfloor D \rfloor - \lfloor U \rfloor \\ &> \lfloor D - U \rfloor. \end{aligned}$$

The set $S = D - U$ is disjoint from all unstable sets and contains more variables than its minimal capacity. It is therefore a stable set. Thus there is always a stable and an unstable set that forms a bipartition of D. ■

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 $\lfloor L \rfloor + \lfloor R \rfloor = I(L \cup R)$.

$$
\begin{aligned}
\lceil F \rceil + \lfloor L \rfloor + \lfloor R \rfloor &\leq \lceil F \rceil + \lceil L \rceil + \lfloor R \rfloor \\
\lceil F \rceil + I(L \cup R) &< C(E) + \lfloor R \rfloor \\
\lceil F \rceil + I(L \cup R) &< |\{x \in X \mid dom(x) \subseteq E \wedge dom(x) \not\subseteq F\}| + C(F) + \lfloor R \rfloor \\
\lceil F \rceil + I(L \cup R) &< |\{x \in X \mid dom(x) \cap L \neq \emptyset \wedge dom(x) \cap R = \emptyset\}| + C(F) + \lfloor R \rfloor \\
\lceil F \rceil + I(R) &< C(F) + \lfloor R \rfloor \\
\lceil F \rceil &< C(F)
\end{aligned}
$$

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 $\lceil F \rceil + \lceil L \rceil = C(H) = |\{x \in X \mid dom(x) \subseteq H \wedge dom(x) \nsubseteq F\}| + C(F)$. Therefore,

$$\lceil F \rceil + \lfloor L \rfloor + \lfloor R \rfloor \leq \lceil F \rceil + \lceil L \rceil + \lfloor R \rfloor$$

$$\lceil F \rceil + I(L \cup R) \leq |\{x \in X \mid dom(x) \subseteq H \wedge dom(x) \nsubseteq F\}| + C(F) + \lfloor R \rfloor$$

$$\lceil F \rceil + I(L \cup R) \leq |\{x \in X \mid dom(x) \cap L \neq \emptyset \wedge dom(x) \cap R = \emptyset\}| + C(F) + \lfloor R \rfloor$$

$$\lceil F \rceil + I(R) \leq C(F) + \lfloor R \rfloor$$

By Theorem 1 all 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., $C(F) = \lceil F \rceil$. Using this observation, we now show that all variables whose domains 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 whose domains are contained in S, algorithm $\mathcal{A}$ already identified F as a Hall set and made all variables consistent with it. Since the variables whose domains are contained in S were not modified by $\mathcal{B}$ they are still consistent with F. A variable whose domain intersects an unstable set like U and R must have both bounds in this set. Since $U = L \cup R$, a variable whose domain intersects U must have both bounds in either L or R and therefore be consistent with the Hall set H. Similarly, one can show the result also holds for domain consistency.

We have shown that any variable whose domain is either contained in S or intersects U is 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. 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 [9] 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. The algorithm time complexity is $O(t + |X|)$ where t is the time required for sorting the variable domains by lower and upper bounds.

4.2. The Lower Bound Constraint (lbc)

We now present an algorithm (see Figure 1) that shrinks the lower bounds of the variable domains received as input. The upper bounds can be updated symmetrically by a similar algorithm and consequently make the *lbc* bounds consistent.

The initialization step assigns to each value $v \in D$ exactly l_v empty *buckets* corresponding to the minimal capacity to be filled for v and setting a *failure flag* which indicates if v belongs to a failure set. The union-find 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 then S is fully contained in this stable set. Stable sets are stored in the variable $Stable$.

Our algorithm processes each variable $x \in X$ in nondecreasing order by upper bound. Like the algorithm of Lipski et al. [8], 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 $I = [\min(dom(x)), v]$ is contained in this stable set. The algorithm regroups all values in I in its variable *PS*. 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*.

The algorithm initially assumes that all values belong to a failure set. When processing variable x, an interval $I = [a, \max(dom(x))]$ with no empty buckets contains the domains of a least $\lfloor I \rfloor$ variables and thus cannot be a failure set. The algorithm unsets the failure flags for all values in I. If a value still has a failure flag set after processing all the variables then the *lbc* is unsatisfiable.

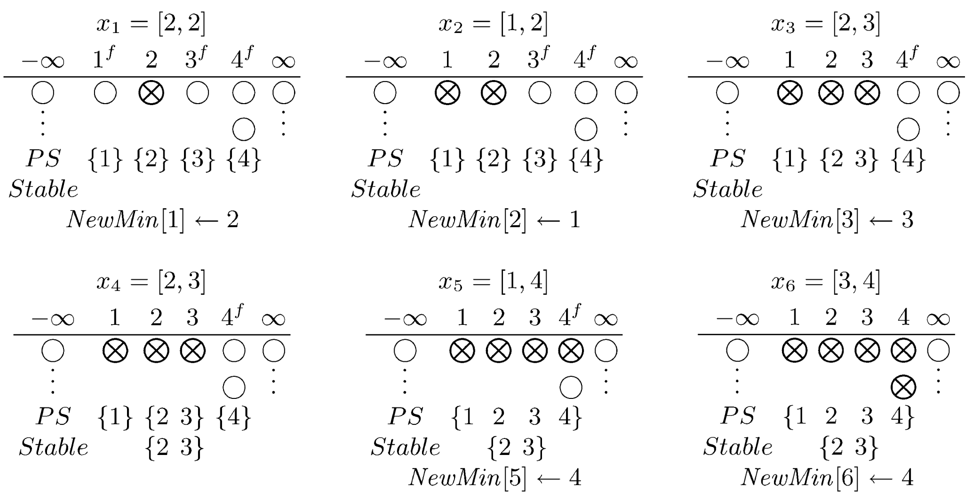

Figure 1. Trace of Algorithm 1.

Algorithm 1: Bounds consistency algorithm for the *lbc*

Let PS be a union-find data structure over the elements in D;
Let $Stable = \emptyset$;
for $v \in D$ **do**
  associate l_v empty buckets to the value v;
  if $l_v > 0$ **then** mark v as a *failure* element;
$D \leftarrow D \cup \{-\infty, \infty\}$;
associate ∞ buckets to the values $-\infty$ and ∞;
for $x_i \in X$ in nondecreasing order of $\max(dom(x_i))$ **do**
  $a \leftarrow \min(dom(x_i))$; $b \leftarrow \max(dom(x_i))$;
  $z \leftarrow \min(\{v \in D \mid v \geq a,\ a$ has an empty bucket$\})$;
  if $z > a$ **then** `union` $(PS, a, a+1, \ldots, \min(b, z))$;
  if $z > b$ **then**
    $S \leftarrow$`findSet` (PS, b);
    $Stable \leftarrow Stable \cup \{S\}$;
  else
    add a token in one of the empty buckets of z;
    $z \leftarrow \min(\{v \in D \mid v \geq a,\ a$ has an empty bucket$\})$;
    $NewMin[i] \leftarrow \min(\{v \in D \mid v \geq a, v$ has a failure flag$\})$;
    if $z > b$ **then**
      $j \leftarrow \max(\{v \in D \mid v \leq b, v$ has an empty bucket$\})$;
      reset the *failure* flag for all elements in $(j, b]$;
if $|\{v \in D \mid v\ has\ a\ failure\ flag\}| > 0$ **then** **return** *Failure*;
for $x_i \in X$ *such that* $\forall S \in Stable,\ dom(x_i) \not\subseteq S$ **do**
  $dom(x_i) \leftarrow dom(x_i) - [\min(dom(x_i)),\ NewMin[i]]$;
return *Success*;

To shrink the domains, the algorithm stores in *NewMin*[*i*] the smallest value $v \in dom(x_i)$ with a failure flag. If $dom(x_i)$ intersected an unstable set U, v would be the smallest value in $dom(x_i) \cap U$. If no values in $dom(x_i)$ have a failure flag, x_i belongs to a stable set and *NewMin*[*i*] remains undefined. After processing all variables, the algorithm assigns the new lower bound *NewMin* to the variables that are not contained in a stable set.

Correctness

We wish to show that the algorithm returns *Success* if and only if the *lbc* has a solution. From the construction of the algorithm's solution it follow trivially that it satisfies the *lbc* constraint.

For the converse, first we observe that a satisfiable *lbc* constraint remains satisfiable if we enlarge the domain of any one variable, as the solution to the original *lbc* is also a solution to the enlarged *lbc*.

Now assume that *lbc* has a solution L; that is, L is a set of assignments of values to variables that satisfy the *lbc* constraint. We compare L with the assignment computed by the algorithm as it proceeds by order of upper bound. The algorithm processes x_1, then x_2 and so on. Every time the algorithm makes an assignment (i.e., places a token in a bucket) we compare to see if L assigns the same value to this variable, until we are at variable x_i which is assigned a value v_i by the algorithm, but is assigned to variable x_j by L. Now, since the algorithm processed x_i before x_j we know that $\max(dom(x_j)) \geq \max(dom(x_i))$.

Hence, L assigns v_i to x_j and assigns x_i a larger value at a later time. The algorithm instead assigns v_i to x_i and uses x_j later. But since $\max(dom(x_j)) \geq \max(dom(x_i))$, it follows that the remaining *lbc* is also satisfiable as enlarging the domain leaves the *lbc* solvable. We now rename the variable x_j to x_i and the algorithm continues. This situation is repeated for any other variables which are assigned differently by the algorithm and L, until all variables are assigned and hence our algorithm finds a solution if one exists.

Lastly, we shrink the domain of variables that intersect an unstable set. Recall that, by Lemma 3, variables that intersect an unstable set cannot be assigned to values outside this set. When we process a variable, we assume that it intersects an unstable set and compute the new lower bound of the variable domain. All variables that have their failure flag unset at the time of processing of the variable already belong to a set S that contains at least as many variables as its minimal capacity; i.e., $C(S) \geq \lfloor S \rfloor$. Hence, if the algorithm processes a variable x that intersects such a set, it is clear that S is not an unstable set and that x is not required by S to satisfy the *lbc*. We therefore store in *NewMin*[x] the first element in *dom*(x) that still has its failure flag set. Later on we test to see if this variable now intersects an unstable set U and must be shrunk.

Example 2. Figure 1 shows 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 $x_6 \in [4, 4]$.

A naive implementation of our algorithm has time complexity $O(t + |X||D|)$, where t is the complexity of sorting the intervals by upper bounds. Incremental and linear time sorting algorithms have time complexity less than $O(|X| \log |X|)$. We will show how to improve the complexity to $O(t + |X|)$.

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. The size of D' is bounded by $2|X|$. 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 $O(t + |X|^2)$.

To get a linear complexity, we implement the buckets using a union-find data structure and an array of integers that stores the number of empty buckets a value v has. If all buckets of a value v are filled in, 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 [3]. The algorithm takes $O(t + |X|)$ steps using the interval union-find for the failure flags, the stable sets *Stable*, and the potential stable sets *PS*.

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. In our experiments (see Section 6), we use instead the tree data structure described in [9] to obtain an algorithm with $O(t + |X|\log|X|)$ time complexity. This tree data structure even offers slightly better performance than the standard union-find data structure which runs in $O(t + |X|\alpha(|X|))$ where α is the inverse of Ackermann's function.

5. Domain Consistency

In this section we present a propagator that makes a *gcc* domain consistent. We will use Régin's propagator [12] for the *alldifferent* constraint as a black box that has complexity $O(|X|^{1.5}|D|)$ 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. For instance, the domain of variable x_2 in Example 1 can be represented by $\{1a, 1b, 1c, 2a, 2b, 2c\}$. We apply Régin's propagator with the new domains and then merge back duplicates to their original value. 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 $O(u|X|^{1.5}|D|)$.

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. Consequently, an unstable set in the primal corresponds to a Hall set in the dual. A propagator for the *alldifferent* constraint removes from a domain the values contained in a Hall set only if the domain is not fully contained in this 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 do not belong to this unstable set. This operation is sufficient to make the primal domain consistent. The *alldifferent* propagator would also return a failure if the problem is unsolvable. A failure set in the primal corresponds to a set of values in the dual that contains more variables than values. Such a set makes the dual unsolvable and is detected by the *alldifferent* propagator. ■

We use Régin's propagator to solve the dual problem and then merge back the duplicated values in the domains to their previous value. 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 $l = max_{v \in D} l_v$.

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 $O(|X|^{1.5}|D|)$. 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 $O(|X|^{1.5}|D|)$. For either class, the complexity of the algorithm improves the previous best *gcc* propagator [13] for domain consistency which runs in $O(|X|^2|D|)$.

5.3. Improving the gcc Propagator

In the previous sections, we saw how one can use an *alldifferent* propagator as a black box to enforce domain consistency on the *gcc*. In this section, we show how to implement the black box in order to get a complexity of $O(|X|^{1.5}|D|)$ for any class of problems.

For the *ubc* and *lbc* problems, we will need to construct a special graph. Following Régin [12], let $G(\langle X, D\rangle, E)$ be an undirected bipartite graph such that nodes at the left represent variables and nodes at the right represent values. There is an edge (x_i, v) in E iff the value v is in the domain $dom(x_i)$ of the variable. Let $c(n)$ be the capacity associated to node n such that $c(x_i) = 1$ for all variable-nodes $x_i \in X$ and $c(v)$ is an arbitrary non-negative value for all value-nodes v in D. A matching M in graph G is a subset of the edges E such that no more than $c(n)$ edges in M are adjacent to node n. We are interested in finding a matching M with maximal cardinality.

The following concepts from flow and matching theory (see [1]) will be useful in this context. Consider a graph G and a matching M. The *residual graph* G_M of G is the directed version of graph G such that edges in M are oriented from values to variables and edges in $E - M$ are oriented from variables to values. A node n is *free* if the number of edges adjacent to n in M is strictly less than its capacity $c(n)$. An *augmenting path* in G_M is a path with an odd number of links that connects two free nodes together. If there is an augmenting path p in G_M, then there exists a matching M' of cardinality $|M'| = |M| + 1$ that is obtained by computing the symmetric difference $M \oplus p$. A matching M is maximal iff there is no augmenting path in the graph G_M.

Hopcroft and Karp [5] describe an algorithm with running time $O(|X|^{1.5}|D|)$ that finds a maximum matching in a bipartite graph when the capacities $c(n)$ are equal to 1 for all nodes. We generalize the algorithm to obtain the same complexity when $c(v) \geq 0$ for the value-nodes and $c(x_i) = 1$ for variable-nodes.

The Hopcroft–Karp algorithm starts with an initial empty matching $M = \emptyset$ which is improved at each iteration by finding a set of disjoint shortest augmenting paths. An iteration that finds a set of augmenting paths proceeds in two steps.

The first step consists of performing a breadth-first search [17] (BFS) on the residual graph G_M starting with the free variable-nodes. The breadth-first search generates a forest of nodes such that nodes at level i are at distance i from a free node. This distance is minimal by property of BFS. Let m be the smallest level that contains a free value-node. For each node n at level $i < m$, we assign a list $L(n)$ of nodes adjacent to node n that are at level $i + 1$. We set $L(n) = \emptyset$ for every node at level m or higher.

The second step of the algorithm uses a stack to perform a depth-first search [17] (DFS). The DFS starts from a free variable-node and is only allowed to branch from a node n to a node in $L(n)$. When the algorithm branches from node n_1 to n_2, it deletes n_2 from $L(n_1)$. If the DFS reaches a free value-node, the algorithm marks this node as non-free, clears the stack, and pushes a new free variable-node that has not been visited onto the stack. This DFS generates a forest of trees whose roots are free variable-nodes. If a tree also contains a free value-node, then the path from the root to this free-value node is an augmenting path. Changing the orientation of all edges that lie on the augmenting paths generates a matching of greater cardinality.

In our case, to find a matching when capacities of value-nodes $c(v)$ are non-negative, we construct the duplicated graph G' where value-nodes v are duplicated $c(v)$ times and the capacity of each node is set to 1. Clearly, a matching in G' corresponds to a matching in G and can be found by the Hopcroft–Karp algorithm. We can simulate a trace of the Hopcroft–Karp algorithm run on graph G' by directly using graph G. We simply let the DFS visit $c(n) - \deg_M(n)$ times a free-node n where $\deg_M(n)$ is the number of edges in M adjacent to node n. This simulates the visit of the free duplicated nodes of node n in G. Even if we allow multiple visits of a same node, we maintain the constraint that an edge cannot be traversed more than once in the DFS. The running time complexity for a DFS is still bounded by the number of edges $O(|X||D|)$.

Hopcroft and Karp proved that if s is the cardinality of a maximum cardinality matching, then $O(\sqrt{s})$ iterations are sufficient to find this maximum cardinality matching. In our case, s is bounded by $|X|$ and the complexity of each BFS and DFS is bounded by the number of edges in G_M i.e., $O(|X||D|)$. The total complexity is therefore $O(|X|^{1.5}|D|)$. We will run this algorithm twice, first with $c(v) = u_v$ to obtain a matching M_u and then with $c(v) = l_v$ to obtain a matching M_l.

5.4. *Pruning the Domains*

Using the algorithm described in Section 5.3, we compute a matching M_u in graph G such that capacities of variable-nodes are set to $c(x_i) = 1$ and capacities of value-nodes are set to $c(v) = u_v$. A matching M_u clearly corresponds to an assignment that satisfies the *ubc* if it has cardinality $|X|$ i.e., if each variable is assigned to a value.

Consider now the same graph G where capacities of variable-nodes are $c(x_i) = 1$ but capacities of value-nodes are set to $c(v) = l_v$. A maximum matching M_l of cardinality $|M_l| = \sum l_v$ represents a partial solution that satisfies the *lbc*. Variables that are not assigned to a value can in fact be assigned to any value in their domain and still satisfy the *lbc*.

Pruning the domains consists of finding the edges that cannot be part of a matching. From flow theory, we know that an edge can be part of a matching iff it belongs to a strongly connected component of the residual graph or lies on a path starting from or leading to a free node.

Régin's algorithm prunes the domains by finding all strongly connected components and flagging all edges that lie on a path starting or finishing at a free node. This can be done in $O(|X||D|)$ using a DFS as described in [17]. Using Theorem 2 and 3, we remove unsupported edges in G_{Mu} and then in G_{M_l} and therefore enforce domain consistency in $O(|X|^{1.5}|D|)$.

Reusing matchings M_u and M_l, Régin shows how an incremental propagator can maintain domain consistency in $O(|X||D|)$ steps. Incremental algorithms are useful when variable domains are pruned by other constraints and domain consistency needs to be reinforced on the *gcc*. Régin's incremental algorithm can also be used with our algorithm.

Our algorithm offers a better running time complexity. The advantage of our method remains to be evaluated in practice.

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 [6].[1] Following a suggestion by Puget [10] 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 [13] 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 [6, 20].

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 ([10]; 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 $dom(x_i) = [i - n, 0]$, $0 \leq i \leq n$, 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 [18] for more details on the problem). In our experiments, we used ten hard problems that were taken from the SPEC95 floating point,

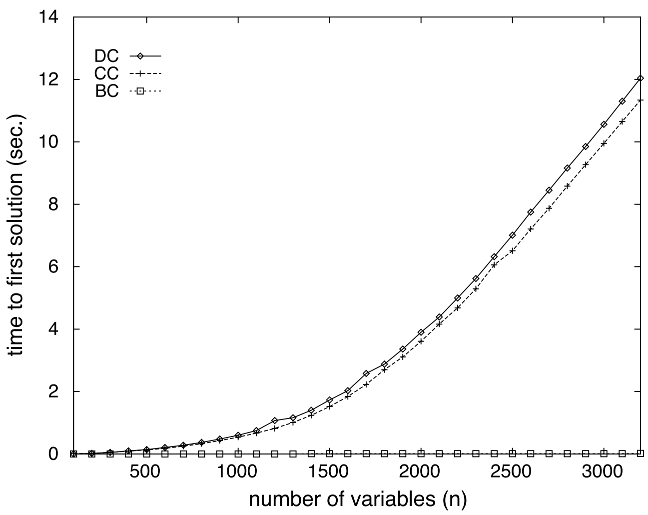

Figure 2. Time (sec.) to first solution for Pathological problems.

Table 1. Time (sec.) to optimal solution for instruction scheduling problems; (left) issue width = 2; (right) 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
1,006			8.70

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			
1,006	307.00		14.44

SPEC2000 floating point, and MediaBench benchmarks. 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 processor with an issue width of two with two identical functional units, and an issue width of four with two floating point units and two integer units (see Table 1). Here, our BC propagator offers a clear performance improvement over the other propagators.

We next consider car sequencing problems (see [6]). 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 [19] 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, 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.

Table 2. (left) Time (sec.) to first solution or to detect inconsistency for car sequencing problems; (right) 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

n	CC	DC	BC	BC+
10	439	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

Table 3. (left) Time (sec.) to first solution for sports league scheduling problems; (right) 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

n	CC	DC	BC	BC+
8	1,308	914	136	942
10	5,767	428	54	689
12	6,449	4,399	149,728	5,356
14	33,901	19,584		22,176

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.

We have also empirically compared our algorithm to Katriel and Thiel's algorithm [7] over the same problems reported above. The implementation of Katriel and Thiel's algorithm was written by those authors. Care was taken to, as much as possible, compare the algorithms rather than the implementations. To this end, both implementations used the same sorting code and the pruning of count variables was disabled in Katriel and Thiel's algorithm. Our algorithm was never slower on the Pathological problems (the maximum speedup of our algorithm over Katriel and Thiel's algorithm was 75%, where the speedup is calculated as the time saved divided by the original time), never slower on the instruction scheduling problems (maximum speedup was 13%), and never slower on the car sequencing problems (maximum speedup was 26%). Katriel and Thiel's algorithm was never slower on the sports league scheduling problems (maximum speedup was 8%) and never slower on the random problems (maximum speedup was 13%).

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 (left) [0, 2]; (right) 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 min. 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
1,600	266.80	1.24

DC			BC		
$d/2$	d	$2d$	$d/2$	d	$2d$
0.00	0.01	0.33	0.00	0.00	0.00
0.00	0.07	4.81	0.00	0.01	0.01
0.01	0.60	74.88	0.00	0.03	0.04
0.03	4.58		0.01	0.15	0.16
0.20	34.78		0.02	0.70	0.62

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.

Acknowledgements

The authors thank the participants of the constraint programming problem session at the University of Waterloo, Kent Wilken for providing the instruction scheduling problems used in our experiments and Irit Katriel and Sven Thiel [7] for trying out our algorithm. Alexander Golynski is partially supported by NSERC grant RGPIN8237 and Claude-Guy Quimper by an NSERC Doctoral Scholarship.

Note

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

References

1. Ahuja, R. K., Magnanti, T. L., & Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications*, 1st Edition. Prentice Hall.
2. Caseau, Y., Guillo, P.-Y., & Levenez, E. (1993). A deductive and object-oriented approach to a complex scheduling problem. In *Deductive and Object-Oriented Databases*, pages 67–80.
3. Gabow, H. N., & Tarjan, R. E. (1983). A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the Fifteenth ACM Symposium on Theory of Computing*, pages 246–251.
4. Hall, P. (1935). On representatives of subsets. *J. Lond. Math. Soc.* 26–30.
5. Hopcroft, J., & Karp, R. (1973). An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.* 2: 225–231.
6. ILOG S. A. (1998). ILOG Solver 4.2 user's manual.
7. Katriel, I., & Thiel, S. (2003). Fast bound consistency for the global cardinality constraint. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*, Kinsale, Ireland, pages 437–451.
8. Lipski, W., & Preparata, F. P. (1981). Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems. *Acta Inform.* 15: 329–346.
9. López-Ortiz, A., Quimper, C.-G., Tromp, J., & van Beek, P. (2003). A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, Acapulco, Mexico, pages 245–250.
10. Puget, J.-F. (1998). A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, Madison, Wisconsin, pages 359–366.
11. Quimper, C.-G., López-Ortiz, A., van Beek, P., & Golynski, A. (2004). Improved algorithms for the global cardinality constraint. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming*, Toronto, Ontario, pages 542–556 (September).

12. Régin, J.-C. (1994). A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, pages 362–367.
13. Régin, J.-C. (1996). Generalized arc consistency for global cardinality constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, Oregon, pages 209–215.
14. Régin, J.-C., & Puget, J.-F. (1997). A filtering algorithm for global sequencing constraints. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, Linz, Austria, pages 32–46.
15. Schulte, C., & Stuckey, P. J. (2001). 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*, Firenze, Italy, pages 115–126.
16. Stergiou, K., & Walsh, T. (1999). The difference all-difference makes. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, Stockholm, pages 414–419.
17. Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM J. Comput.* 1: 146–160.
18. van Beek, P., & Wilken, K. (2001). 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*, Paphos, Cyprus, pages 625–639.
19. Van Hentenryck, P., Michel, L., Perron, L., & Régin, J.-C. (1999). Constraint programming in OPL. In *Proceedings of the First International Conference on Principles and Practice of Declarative Programming*, Paris, pages 98–116.
20. Van Hentenryck, P., Simonis, H., & Dincbas, M. (1992) Constraint satisfaction using constraint logic programming. *Artif. Intell.* 58: 113–159.