Among, Common and Disjoint Constraints

Christian Bessiere[1], Emmanuel Hebrard[2], Brahim Hnich[3], Zeynep Kiziltan[4], and Toby Walsh[2]

[1] LIRMM, CNRS/University of Montpellier, France
bessiere@lirmm.fr
[2] NICTA and UNSW, Sydney, Australia
{ehebrard, tw}@cse.unsw.edu.au
[3] Izmir University of Economics, Turkey
brahim.hnich@ieu.edu.tr
[4] University of Bologna, Italy
zkiziltan@deis.unibo.it

Abstract. Among, Common and Disjoint are global constraints useful in modelling problems involving resources. We study a number of variations of these constraints over integer and set variables. We show how computational complexity can be used to determine whether achieving the highest level of consistency is tractable. For tractable constraints, we present a polynomial propagation algorithm and compare it to logical decompositions with respect to the amount of constraint propagation. For intractable cases, we show in many cases that a propagation algorithm can be adapted from a propagation algorithm of a similar tractable one.

1 Introduction

Global constraints are an essential aspect of constraint programming. See, for example, [8, 3, 9, 2]. They specify patterns that occur in many problems, and exploit efficient and effective propagation algorithms to prune search. In problems involving resources, we often need to constrain the number of variables taking particular values. For instance, we might want to limit the number of night shifts assigned to a given worker, to ensure some workers are common between two shifts, or to prevent any overlap in shifts between workers who dislike each other. The Among, Common and Disjoint constraints respectively are useful in such circumstances. The Among, Common and Disjoint constraints are useful in such circumstances.

The Among constraint was first introduced in CHIP to model resource allocation problems like car sequencing [3]. It counts the number of variables using values from a given set. A generalization of the Among and AllDifferent constraints is the Common constraint [2]. Given two sets of variables, this counts the number in each set which use values from the other set. A special case of the Common constraint also introduced in [2] is the Disjoint constraint. This ensures that no value is common between two sets of variables. We study these three global constraints as well as seven other variations over integer and set variables. For each case, we present a polynomial propagation algorithm, and identify when achieving a higher level of local consistency is intractable. For example,

B. Hnich et al. (Eds.): CSCLP 2005, LNAI 3978, pp. 29–43, 2006.

rather surprisingly, even though the DISJOINT constraint is closely related to (but somewhat weaker than) the ALLDIFFERENT constraint, it is NP-hard to achieve generalised arc consistency on it.

The rest of the paper is oragnised as follows. We first present the necessary formal background in Section 2. Then, in Section 3 and Section 4 we study various generalisations and specialisations of the AMONG, COMMON, and DISJOINT constraints on integer and set variables. Finally, we review related work in Section 5 before we conclude and present our future plans in Section 6.

2 Formal Background

A constraint satisfaction problem consists of a set of variables, each with a finite domain of values, and a set of constraints specifying allowed combinations of values for given subsets of variables. A solution is an assignment of values to the variables satisfying the constraints. We consider both integer and set variables. A set variable S can be represented by a lower bound $lb(S)$ which contains the definite elements and an upper bound $ub(S)$ which contains the definite and potential elements. We use the following notations: X, Y, N, and M (possibly with subscripts) denote integer variables; S and T (again possibly with subscripts) denote set variables; $\mathcal{S}$ (possibly with a subscript) and $\mathcal{K}$ denote sets of integers; and v and k (possibly with a subscript) denote integer values. We write $\mathcal{D}(X)$ for the domain of a variable X. For integer domains, we write $min(X)$ and $max(X)$ for the minimum and maximum elements in $\mathcal{D}(X)$. Throughout the paper, we consider constraint satisfaction problems in which a constraint contains no repeated variables.

Constraint solvers often search in a space of partial assignments enforcing a local consistency property. A *bound support* for a constraint C is a partial assignment which satisfies C and assigns to each integer variable in C a value between its minimum and maximum, and to each set variable in C a set between its lower and upper bounds. A bound support in which each integer variable takes a value in its domain is a *hybrid support*. If C involves only integer variables, a hybrid support is a *support*. A constraint C is *bound consistent* (BC) iff for each integer variable X, $min(X)$ and $max(X)$ belong to a bound support, and for each set variable S, the values in $ub(S)$ belong to S in at least one bound support and the values in $lb(S)$ are those from $ub(S)$ that belong to S in all bound supports. A constraint C is *hybrid consistent* (HC) iff for each integer variable X, every value in $D(X)$ belongs to a hybrid support, and for each set variable S, the values in $ub(S)$ belong to S in at least one hybrid support and the values in $lb(S)$ are those from $ub(S)$ that belong to S in all hybrid supports. A constraint C over integer variables is *generalized arc consistent* (GAC) iff for each variable X, every value in $D(X)$ belongs to a support. If all variables in C are integer variables, HC is equivalent to GAC, whilst if all variables in C are set variables, HC is equivalent to BC. Finally, we will compare local consistency properties applied to (sets of) logically equivalent constraints. A local consistency property Φ on C_1 is *strictly stronger* than Ψ on C_2 iff, given any domains, Φ removes all values Ψ removes, and sometimes more.

3 Integer Variables

3.1 Among Constraint

The AMONG constraint counts the number of variables using values from a given set [3]. More formally, we have:

$$\text{AMONG}([X_1, .., X_n], [k_1, .., k_m], N) \text{ iff } N = |\{i \mid \exists j \ . \ X_i = k_j\}|$$

For instance, we can use this constraint to limit the number of tasks (variables) assigned to a particular resource (value). Enforcing GAC on such a constraint is polynomial. Before we give an algorithm to do this, we establish the following theoretical results.

Lemma 1. *Given* $\mathcal{K} = \{k_1, .., k_m\}$, $lb = |\{i \mid \mathcal{D}(X_i) \subseteq \mathcal{K}\}|$, *and* $ub = n - |\{i \mid \mathcal{D}(X_i) \cap \mathcal{K} = \emptyset\}|$, *a value* $v \in \mathcal{D}(N)$ *is GAC for* AMONG *iff* $lb \leq v \leq ub$.

Proof. At most ub variables in $[X_1, .., X_n]$ can take a value from $\mathcal{K}$ and lb of these take values only from $\mathcal{K}$. Hence v is inconsistent if $v < lb$ or $v > ub$. We now need to show any value between lb and ub is consistent. We have $ub - lb$ variables that can take a value from $\mathcal{K}$ as well from outside $\mathcal{K}$. A support for $lb \leq v \leq ub$ can be constructed by assigning v variables to a value from $\mathcal{K}$ and $ub - v$ variables to a value from outside $\mathcal{K}$. □

Lemma 2. *Given* $\mathcal{K} = \{k_1, .., k_m\}$, $lb = |\{i \mid D(X_i) \subseteq \mathcal{K}\}|$, $ub = n - |\{i \mid D(X_i) \cap \mathcal{K} = \emptyset\}|$, *and* $lb \leq min(N) \leq max(N) \leq ub$, *a value in* $D(X_i)$ *may not be GAC for* AMONG *iff* $lb = min(N) = max(N)$ *or* $min(N) = max(N) = ub$.

Proof. The variables $[X_1, .., X_n]$ can be divided into three categories: 1) those whose domain contains values only from $\mathcal{K}$ (lb of them), 2) those whose domain contains both values from $\mathcal{K}$ and from outside ($ub - lb$ of them), and 3) those whose domain does not intersect with $\mathcal{K}$ ($n - ub$ of them). If $lb = min(N) = max(N)$ then exactly lb variables must take a value from $\mathcal{K}$. These variables can then only be those of the first category and thus $\mathcal{K}$ cannot be in the domains of the second category. If $min(N) = max(N) = ub$ then exactly ub variables must take a value from $\mathcal{K}$. These variables can then only be those of the first and the second category and thus any value $v \notin \mathcal{K}$ cannot be in the domains of the second category. We now need to show this is the only possibility for inconsistency. Consider an assignment to the constraint. Due to the variables of the first and the third category we have lb values from $\mathcal{K}$ and $n - ub$ values from outside $\mathcal{K}$. If $lb < max(N)$ then in the second category we can have at least one variable assigned to a value from $\mathcal{K}$, the rest assigned to a value outside $\mathcal{K}$ and satisfy the constraint. Similarly, if $min(N) < ub$ then in the second category we can have at least one variable assigned to a value outside of $\mathcal{K}$, the rest assigned to a value from $\mathcal{K}$ and satisfy the constraint. Hence, all values are consistent when $lb < max(N)$ or $min(N) < ub$. □

We now give an algorithm for the AMONG constraint.

Algorithm 1. GAC for $\textsc{Among}([X_1, .., X_n], \mathcal{K}, N)$.

1 $lb := |\{i \mid D(X_i) \subseteq \mathcal{K}\}|$;
2 $ub := n - |\{i \mid D(X_i) \cap \mathcal{K} = \emptyset\}|$;
3 $min(N) := max(min(N), lb)$;
4 $max(N) := min(max(N), ub)$;
5 **if** $(max(N) < min(N))$ **then** fail;
6 **if** $(lb = min(N) = max(N))$ **then**
  foreach X_i . $\mathcal{D}(X_i) \not\subseteq \mathcal{K}$ **do** $\mathcal{D}(X_i) := \mathcal{D}(X_i) \setminus \mathcal{K}$;
7 **if** $(min(N) = max(N) = ub)$ **then**
  foreach X_i . $\mathcal{D}(X_i) \cap \mathcal{K} \neq \emptyset$ **do** $\mathcal{D}(X_i) := \mathcal{D}(X_i) \cap \mathcal{K}$;

Theorem 1. *Algorithm 1 maintains GAC on* $\textsc{Among}([X_1, .., X_n], [k_1, .., k_m], N)$ *and runs in* $O(nd)$ *where* d *is the maximum domain size.*

Proof. (Sketch) By Lemmas 1 and 2, the algorithm maintains GAC. Computing lb and ub is in $O(nd)$. Updating the bounds of N is constant time. Updating $\mathcal{D}(X_i)$ is in $O(d)$. Since there are n variables, pruning X_i's is in $O(nd)$. Thus, GAC on Among is in $O(nd)$. □

The behaviour of the algorithm can be simulated by encoding the Among constraint using the sum constraint:

$$\textsc{Among}([X_1, \ldots, X_n], \mathcal{K}, N) \text{ iff}$$
$$\forall i \in \{1, .., n\}\ B_i = 1 \leftrightarrow X_i \in \mathcal{K} \wedge \sum_{i \in \{1,..,n\}} B_i = N$$

where each B_i is a Boolean variable with the domain $\{0, 1\}$. In the algorithm, lb corresponds to the number of Boolean variables assigned 1, and ub to the number of Boolean variables not assigned 0 (that is, either assigned 1 or having the domain {0,1}). Lines 3 and 4 of the algorithm can be seen as the propagation of the sum constraint: $min(N)$ is computed by taking the maximum of $min(N)$ and the sum of $min(B_i)$ which is equivalent to lb; similarly $max(N)$ is computed by taking the minimum of $max(N)$ and the sum of $max(B_i)$ which is equivalent to ub. If $lb = min(N) = max(N)$, all the Booleans having the $\{0, 1\}$ domain will be assigned 0, meaning that the associated variables do not take values from $\mathcal{K}$. Likewise, if $min(N) = max(N) = ub$, all the Booleans having the $\{0, 1\}$ domain will be assigned 1, meaning that the associated variables take values only from $\mathcal{K}$. Otherwise, no propagation will occur. Consequently, the sum decomposition maintains GAC.

An alternative method of propagating an Among constraint is using the global cardinality constraint Gcc [9]:

$$\textsc{Among}([X_1, \ldots, X_n], [k_1, .., k_m], N) \text{ iff}$$
$$\textsc{Gcc}([X_1, \ldots, X_n], [k_1, .., k_m], [O_1, .., O_m]) \ \wedge$$
$$\sum_{i \in \{1,..,m\}} O_i = N$$

As shown in [6], this decomposition may not always achieve GAC.

Even if GAC on AMONG can be maintained by a simple decomposition, the presented algorithm is useful when we consider a number of extensions of the AMONG constraint. An interesting extension is when we count not the variables taking some given values but those taking values taken by other variables. This is useful when, for example, the resources to be used are not initially known. We consider here two such extensions in which we replace $[k_1, .., k_m]$ either by a set variable S or by a sequence of variables $[Y_1, .., Y_m]$

AMONG$([X_1, .., X_n], S, N)$ holds iff N variables in X_i take values in the set S. That is, $N = |\{i \mid X_i \in S\}|$. Enforcing HC on this constraint is NP-hard in general.

Theorem 2. *Enforcing HC on* AMONG$([X_1, .., X_n], S, N)$ *is NP-hard.*

Proof. We reduce 3-SAT to the problem of deciding if such an AMONG constraint has a satisfying assignment. Finding hybrid support is therefore NP-hard. Consider a formula φ with n variables (labelled from 1 to n) and m clauses. Let k be $m + n + 1$. To construct the AMONG constraint, we create $2k + 1$ variables for each literal i in the formula such that $X_{i1}..X_{ik} \in \{i\}$, $X_{i(k+1)}..X_{i(2k)} \in \{-i\}$, and $X_{i(2k+1)} \in \{i, -i\}$. We create a variable Y_j for each clause j in φ and let $Y_j \in \{x, -y, z\}$ where the jth clause in φ is $x \vee \neg y \vee z$. We let $N = n(k + 1) + m$ and $\{\} \subseteq S \subseteq \{1, -1, .., n, -n\}$. The constraint AMONG$([X_{11}, .., X_{1(2k+1)}, .., X_{n1}, .., X_{n(2k+1)}, Y_1, .., Y_m], S, N)$ has a solution iff φ has a satisfying assignment. ♡

In Algorithm 2, we give a propagation algorithm for this AMONG constraint. Notice that we assume all values are strictly positive. We highlight the differences with Algorithm 1. The first modification is to replace each occurrence of $\mathcal{K}$ by either $lb(S)$ or $ub(S)$. As a consequence, instead of a single lower bound and upper bound on N, we have now two pairs of bounds, one under the hypothesis that S is fixed to its lower bound ($lb[0]$ and $glb[0]$), and one under the hypothesis that S is fixed to its upper bound ($lub[0]$ and $ub[0]$). Moreover, in loop 1, we compute the contingent values of lb (resp. ub) when a value v is added to $lb(S)$ (resp. removed from $ub(S)$) and store the results in $lb[v]$ (resp. $ub[v]$). These arrays are necessary for pruning N (lines 3, 4, 6, 7), when the minimum (resp. maximum) value of N cannot be achieved with the current lower (resp. upper) bound of S (conditionals 2 and 5). In this case, we know that at least one of these values must be added to $lb(S)$ (resp. removed from $ub(S)$). Therefore the smallest value $lb[v]$ (resp. greatest value $ub[v]$) is a valid lower bound (resp. upper bound) on N. We also use them for pruning S (lines 8 and 9). Finally, we need to compute lb and ub, as they may have been affected by the pruning on S. This is done in line 10. The worst case time complexity is unchanged, as loop 1 can be done in $O(nd)$.

The level of consistency achieved by this propagation algorithm is incomparable to BC. The following example shows that BC is not stronger: $X_1 \in \{2, 3\}$, $X_2 \in \{2, 3\}$, $X_3 \in \{1, 2, 3, 4\}$, $lb(S) = ub(S) = \{2, 3\}$, $min(N) = max(N) = 2$. The algorithm will prune $\{2, 3\}$ from X_3, whereas a BC algorithm will not do

Algorithm 2. Propagation for $\textsc{Among}([X_1, .., X_n], S, N)$.

$lb[0] := |\{X_i \mid D(X_i) \subseteq lb(S)\}|;$
$glb[0] := n - |\{X_i \mid D(X_i) \cap lb(S) = \emptyset\}|;$
$ub[0] := n - |\{X_i \mid D(X_i) \cap ub(S) = \emptyset\}|;$
$lub[0] := |\{X_i \mid D(X_i) \subseteq ub(S)\}|;$
1 **foreach** $v \in ub(S) \setminus lb(s)$ **do**
  $lb[v] := |\{X_i \mid D(X_i) \subseteq (lb(S) \cup \{v\})\}|;$
  $ub[v] := n - |\{X_i \mid D(X_i) \cap (ub(S) \setminus \{v\}) = \emptyset\}|;$
2 **if** $glb[0] < min(N)$ **then**
3 $LB := \{lb[v] \mid v \in (ub(S) \setminus lb(S))\};$
4 **if** $(LB \not= \emptyset)$ **then** $min(N) = min(LB);$
  else
  $min(N) := max(min(N), lb[0]);$
5 **if** $lub[0] > max(N)$ **then**
6 $UB := \{ub[v] \mid v \in (ub(S) \setminus lb(S))\};$
7 **if** $(UB \not= \emptyset)$ **then** $max(N) = max(UB);$
  else
  $max(N) := min(max(N), ub[0]);$
  if $(max(N) < min(N))$ **then** fail;
8 $lb(S) := lb(S) \cup \{v \mid ub[v] < min(N)\};$
9 $ub(S) := ub(S) \setminus \{v \mid lb[v] > max(N)\};$
10 **if** $(min(N) = max(N))$ **then**
  $lb := |\{i \mid \mathcal{D}(X_I) \subseteq lb(S)\}|;$
  $ub := |\{i \mid \mathcal{D}(X_I) \cap ub(S) \not= \emptyset\}|;$
  if $(lb = min(N))$ **then**
  foreach X_i . $\mathcal{D}(X_i) \not\subseteq lb(S)$ **do** $\mathcal{D}(X_i) := \mathcal{D}(X_i) \setminus lb(S);$
  if $(ub = max(N))$ **then**
  foreach X_i . $\mathcal{D}(X_i) \cap ub(S) \not= \emptyset$ **do** $\mathcal{D}(X_i) := \mathcal{D}(X_i) \cap ub(S);$

any pruning. On the other hand, the following example shows that this algorithm does not enforce BC. Consider $X_1 \in \{1, 2\}$, $X_2 \in \{1, 2\}$, $X_3 \in \{3\}$, $X_4 \in \{3\}$, $X_5 \in \{4\}$, $X_6 \in \{4\}$, $X_6 \in \{5\}$, $X_8 \in \{5\}$, $lb(S) = \{1, 2\}$, $ub(S) = \{1, 2, 3, 4, 5\}$, $N \in \{5, 6, 7, 8\}$. The algorithm will not do any pruning whereas a BC algorithm will prune 5 from N.

We can again use the sum constraint to encode the Among constraint:

$$\textsc{Among}([X_1, .., X_n], S, N) \text{ iff}$$
$$\forall i \in \{1, .., n\}\ B_i = 1 \leftrightarrow X_i \in S \wedge \sum_{i \in \{1,..,n\}} B_i = N$$

where each B_i is a Boolean variable with the domain $\{0, 1\}$. Algorithm 2 is strictly stronger than such a decomposition. It is easy to see that whenever the decomposition prunes a value from N, X_i's, or S, our algorithm also can detect these inconsistencies. However, Algorithm 2 might detect more inconsistent values than the decomposition. For instance, consider $X_1 \in \{1, 2\}$, $X_2 \in \{1, 2\}$, $X_3 = 3$, $X_4 = 3$, $X_5 = 4$, $X_6 = 4$, $lb(S) = \{1, 2\}$, $ub(S) = \{1, 2, 3, 4\}$, and $N \in \{2, 3\}$. Algorithm 2 prunes 3 and 4 from $ub(S)$ but the decomposition does not.

The level of consistency achieved by this decomposition is also incomparable to BC. The example which demonstrates the incomparability of Algorithm 2 and BC also shows that the decomposition is incomparable to BC.

It remains an open question, however, whether BC on such a constraint is tractable or not.

We now consider the second generalization. $\textsc{Among}([X_1, .., X_n], [Y_1, .., Y_m], N)$ holds iff N variables in X_i take values in common with Y_j. That is, $N = |\{i \mid \exists j \ . \ X_i = Y_j\}|$. As before, we cannot expect to enforce GAC on this constraint.

Theorem 3. *Enforcing GAC on* $\textsc{Among}([X_1, .., X_n], [Y_1, .., Y_m], N)$ *is NP-hard.*

Proof. We again use a transformation from 3-Sat. Consider a formula φ with n variables (labelled from 1 to n) and m clauses. We construct the constraint $\textsc{Among}([Y_1, .., Y_m], [X_1, .., X_n], M)$ in which X_i represents the variable i and Y_j represents the clause j in φ. We let $M = m$, $X_i \in \{i, -i\}$ and $Y_j \in \{x, -y, z\}$ where the jth clause in φ is $x \vee \neg y \vee z$. The constructed Among constraint has a solution iff φ has a model. ♡

To propagate $\textsc{Among}([X_1, .., X_n], [Y_1, .., Y_m], N)$, we can use the following decomposition:

$$\begin{aligned} &\textsc{Among}([X_1, .., X_n], [Y_1, .., Y_m], N) \ \text{ iff} \\ &\textsc{Among}([X_1, .., X_n], S, N) \wedge \bigcup_{j \in \{1,..,m\}} \{Y_j\} = S \end{aligned}$$

We can therefore use the propagation algorithm proposed for $\textsc{Among}([X_1, .., X_n], S, N)$. However, even if we were able to enforce HC on the decomposition (which is NP-hard in general to do), we may not make the original constraint GAC.

Theorem 4. *GAC on* $\textsc{Among}([X_1, .., X_n], [Y_1, .., Y_m], N)$ *is strictly stronger than HC on the decomposition.*

Proof: It is at least as strong. To show the strictness, consider $Y_1 \in \{1, 2, 3\}$, $X_1 \in \{1, 2\}$, $X_2 \in \{1, 2, 3\}$, $N = 2$. We have $\{\} \subseteq S \subseteq \{1, 2, 3\}$, hence the decomposition is HC. However, enforcing GAC on $\textsc{Among}([X_1, X_2], [Y_1], N)$ prunes 3 from Y_1 and X_2. ♡

Again, we still do not know whether BC on such a constraint is tractable or not.

3.2 Common Constraint

A generalization of the Among and AllDifferent constraints introduced in [2] is the following Common constraint:

$$\textsc{Common}(N, M, [X_1, .., X_n], [Y_1, .., Y_m]) \text{ iff}$$

$$N = |\{i \mid \exists j \ . \ X_i = Y_j\}| \wedge M = |\{j \mid \exists i \ . \ X_i = Y_j\}|$$

That is, N variables in X_i take values in common with Y_j and M variables in Y_j take values in common with X_i. Hence, the AllDifferent constraint is a special case of the Common constraint in which the Y_j enumerate all the values j in X_i, $Y_j = \{j\}$ and $M = n$. Not surprisingly, enforcing GAC on Common is NP-hard in general, as the result immediately follows from the intractability of the related Among constraint.

Theorem 5. *Enforcing GAC on* Common *is NP-hard.*

Proof. Consider the reduction in the proof of Theorem 3. We let $N \in \{1, .., n\}$. The constructed Common constraint has a solution iff the original 3-Sat problem has a model. ♡

As we have a means of propagation for $\text{Among}([X_1, .., X_n], [Y_1, .., Y_m], N)$, we can use it to propagate the Common constraint using the following decomposition:

$$\begin{aligned}&\text{Common}(N, M, [X_1, .., X_n], [Y_1, .., Y_m]) \text{ iff}\\ &\text{Among}([X_1, .., X_n], [Y_1, .., Y_m], N) \ \wedge\\ &\text{Among}([Y_1, .., Y_m], [X_1, .., X_n], M)\end{aligned}$$

In the next theorem, we prove that we might not achieve GAC on Common even if we do so on Among.

Theorem 6. *GAC on* Common *is strictly stronger than GAC on the decomposition.*

Proof: It is at least as strong. To show the strictness, consider $N = 2$, $M = 1$, $X_1, Y_1 \in \{1, 2\}$, $X_2 \in \{1, 3\}$, $Y_2 \in \{1\}$, and $Y_3 \in \{2, 3\}$. The decomposition is GAC. However, enforcing GAC on $\text{Common}(N, M, [X_1, X_2], [Y_1, Y_2, Y_3])$ prunes 2 from X_1, 3 from X_2, and 1 from Y_1. ♡

Similar to the previous cases, the tractability of BC on such a constraint needs further investigation.

3.3 Disjoint Constraint

We may require that two sequences of variables be disjoint (i.e. have no value in common). For instance, we might want the sequence of shifts assigned to one person to be disjoint from those assigned to someone who dislikes them. The $\text{Disjoint}([X_1, .., X_n], [Y_1, .., Y_m])$ constraint introduced in [2] is a special case of the Common constraint where $N = M = 0$. It ensures $X_i \neq Y_j$ for any i and j. Surprisingly, enforcing GAC remains intractable even in this special case.

Theorem 7. *Enforcing GAC on* Disjoint *is NP-hard.*

Proof: We again use a transformation from 3-Sat. Consider a formula φ with n variables (labelled from 1 to n) and m clauses. We construct the Disjoint constraint in which X_i represents the variable i and Y_j represents the clause j in φ.

We let $X_i \in \{i, -i\}$ and $Y_j \in \{-x, y, -z\}$ where the jth clause in φ is $x \vee \neg y \vee z$. The constructed DISJOINT constraint has a solution iff φ has a model. ♡

An obvious decomposition of the DISJOINT constraint is to post an inequality constraint between every pair of X_i and Y_j, for all $i \in \{1, .., n\}$ and for all $j \in \{1, .., m\}$. Not surprisingly, the decomposition hinders propagation (otherwise we would have a polynomial algorithm for a NP-hard problem).

Theorem 8. *GAC on* DISJOINT *is strictly stronger than AC on the binary decomposition.*

Proof: It is at least as strong. To show the strictness, consider $X_1, Y_1 \in \{1, 2\}$, $X_2, Y_2 \in \{1, 3\}, Y_3 \in \{2, 3\}$. Then all the inequality constraints are AC. However, enforcing GAC on DISJOINT$([X_1, X_2], [Y_1, Y_2, Y_3])$ prunes 2 from X_1, 3 from X_2, and 1 from both Y_1 and Y_2. ♡

This decomposition is useful if we want to maintain BC on DISJOINT.

Theorem 9. *BC on* DISJOINT *is equivalent to BC on the decomposition.*

Proof. It is at least as strong. To show the equivalence, we concentrate on X_i's, but the same reasoning applies to Y_j's. Given X_k where $k \in \{1, .., n\}$, we show that for any bound b_k of X_k ($b_k = min(X_k)$ or $b_k = max(X_k)$) there exists a bound support containing it. We partition the integers as follows. S_X contains all integers v such that $\exists X_i,\ D(X_i) = \{v\}$, S_Y contains all integers w such that $\exists Y_j,\ D(Y_j) = \{w\}$, and T contains the remaining integers. T inherits the total ordering on the integers. So, we can partition T in two sets T_1 and T_2 such that no pair of integers consecutive in T belong both to T_1 or both to T_2. T_1 denotes the one containing b_k if $b_k \in T$. The four sets S_X, S_Y, T_1, T_2 all have empty intersections. Hence, if all X_i can take their value in $S_X \cup T_1$ and all Y_j in $S_Y \cup T_2$, we have a bound support for (X_k, b_k) on the DISJOINT constraint. We have to prove that $[min(X_i)..max(X_i)]$ intersects $S_X \cup T_1$ for any $i \in \{1, .., n\}$ (and similarly for Y_j and $S_Y \cup T_2$). Since $X_i \neq Y_j$ is BC for any j, $min(X_i)$ and $max(X_i)$ cannot be in S_Y. If $min(X_i)$ or $max(X_i)$ is in S_X or T_1, we are done. Now, if both $min(X_i)$ and $max(X_i)$ are in T_2, this means that there is a value between $min(X_i)$ and $max(X_i)$, which is in T_1, by construction of T_1 and T_2. As a result, any bound is BC on DISJOINT if the decomposition is BC. ♡

From Theorem 9, we deduce that BC can be achieved on DISJOINT in polynomial time. In fact, we can achieve more than BC in polynomial time.

Theorem 10. *AC on the binary decomposition is strictly stronger than BC on* DISJOINT.

Proof. AC on the decomposition is at least as strong BC on the decomposition which is equivalent to BC on the original constraint. The following example shows strictness. Consider $X_1 \in \{1, 2, 3\}$ and $Y_1 \in \{2\}$. The constraint DISJOINT$([X_1], [Y_1])$ is BC whereas GAC on the decomposition prunes 2 from X_1. ♡

Algorithm 3. BC for $\textsc{Among}([S_1, .., S_n], \mathcal{K}, N)$.

```
1 InLb := f([lb(S_1, .., lb(S_n))], K);
2 InUb := f([ub(S_1, .., ub(S_n))], K);
3 min(N) := max(min(N), InLb);
4 max(N) := min(max(N), InUb);
5 if min(N) > max(N) then fail;
6 if max(N) = InLb then
      foreach S_i . lb(S_i) ∩ K = ∅ do  ub(S_i) := ub(S_i) \ K;
7 if min(N) = InUb then
      foreach S_i . lb(S_i) ∩ K = ∅ ∧ |K ∩ ub(S_i)| = 1 do
          lb(S_i) := lb(S_i) ∪ K ∩ ub(S_i);
```

4 Set Variables

Many problems involve finding a set of values (for example, the set of nurses on a particular shift). It is useful therefore to have global constraints over set variables [10]. For instance, we might want to count the number of times each nurse has a shift during the monthly roster where each shift is a set variable listing the nurses on duty. This could be achieved with a global constraint that counted the values occurring in a sequence of set variables.

4.1 Among Constraint

We consider an Among constraint over set variables that counts the number of these variables which contain one of the given values. More formally, we have:

$$\textsc{Among}([S_1, .., S_n], [k_1, .., k_m], N) \text{ iff } N = |\{i \mid \exists j \ . \ k_j \in S_i\}|$$

Enforcing BC on such a constraint is polynomial. We propose an algorithm to do this where we define the function $f([\mathcal{S}_1, .., \mathcal{S}_n], \mathcal{K})$ to be $|\{i \mid \mathcal{S}_i \cap \mathcal{K} \neq \emptyset\}|$.

Theorem 11. *Algorithm 3 maintains BC on* $\textsc{Among}([S_1, .., S_n], \mathcal{K}, N)$ *and runs in* $O(nd)$ *where* d *is the size of the maximum upper bound of the* S_i .

Proof. Soundness is relatively immediate. N cannot be greater than the number of variables having k_j's in their upper bounds or smaller than the number of variables having k_j's in their lower bounds. Furthermore, if $max(N)$ is equal to the number of variables having k_j's in their lower bound, there is no hope to satisfy the Among constraint if we use a value k_j in another variable. If $min(N) = InUb$, then for each S_i, if there exists only one element in its upper bound (but not in its lower bound) which is also in $\mathcal{K}$, then that element has necessarily to belong to the lower bound of S_i as it cannot be covered by another S_j otherwise $min(N) < InUb$. Finally, if $min(N) > max(N)$ we necessarily fail.

To show completeness, we need that when we do not fail, the domains returned are bound consistent. Consider an integer k such that $InLb \leq k \leq InUb$. We can construct an assignment of S_i's where exactly k of them take a value in $\mathcal{K}$. We

first assign all S_i's with their lower bound. $InLb$ of the S_i's necessarily contain some k_j since their lower bound overlaps $\mathcal{K}$. For $k - InLb$ variables among the $InUb - InLb$ variables with some k_j in their upper bound but none in their lower bound, we take some k_j from their upper bound to obtain a satisfying assignment with $N = k$. Since $min(N) \geq InLb$ and $max(N) \leq InUb$ (lines 3 and 4), N is BC. Suppose now a value v in $ub(S_i)$. The only case in which v should not belong to $ub(S_i)$ is when v is in the k_j's, none of the k_j's appear in $lb(S_i)$, and no more variable can take values in the k_j's, i.e., $InLb = max(N)$. Then, v will have been removed from $ub(S_i)$ (line 6). In addition, suppose v should belong to $lb(S_i)$. This is the case only if there is no k_j in $lb(S_i)$, and v is the only value in $ub(S_i)$ appearing in the k_j's, and $InUb = min(N)$. Then, v will have been added in $lb(S_i)$ (line 7).

Computing the counters $InLb$ and $InUb$ is in $O(nd)$. Updating the bounds on N is constant time. Deleting values that are not bound consistent in a $ub(S_i)$ or adding a value in $lb(S_i)$ is in $O(d)$. Since there are n variables, this phase is again in $O(nd)$. Bound consistency on Among is in $O(nd)$. ♡

Note we can also add non-empty or cardinality conditions to the S_i without making constraint propagation intractable.

We again consider an extension in which we replace $[k_1, .., k_m]$ by a set variable S. Unlike the previous Among constraint, enforcing BC on Among$([S_1, .., S_n], S, N)$ is NP-hard in general.

Theorem 12. *Enforcing BC on* Among$([S_1, .., S_n], S, N)$ *is NP-hard.*

Proof. We reuse the reduction from the proof of Theorem 2 with minor modifications. We create $2k + 1$ set variables for each literal i in the formula such that $S_{i1}..S_{ik} \in \{i\}..\{i\}$, $S_{i(k+1)}..S_{i(2k)} \in \{-i\}..\{-i\}$, and $S_{i(2k+1)} \in \{\}..\{i, -i\}$. We create a set variable T_j for each clause j in φ and let $T_j \in \{\}..\{x, -y, z\}$ where the jth clause in φ is $x \vee \neg y \vee z$. We let $N = n(k + 1) + m$ and $\{\} \subseteq S \subseteq \{1, -1, .., n, -n\}$. The constraint Among$([S_{11}, .., S_{1(2k+1)}, .., S_{n1}, .., S_{n(2k+1)}, T_1, .., T_m], S, N)$ has a solution iff φ has a model. ♡

Note that the constraint remains intractable if the S_i are non-empty or have a fixed cardinality. We can easily modify the reduction by adding distinct "dummy" values to S_i and T_j respectively. We also add these dummy values to the lower bound of S.

Despite this intractability result, we can easily modify Algorithm 3 to derive a filtering algorithm for Among$([S_1, .., S_n], S, N)$ without changing the complexity. We use the lower bound of S (resp. $ub(S)$) in the computation of $InLb$ (resp. $InUb$). Also, instead of $\mathcal{K}$ in line 6 (resp. line 7), we use $lb(S)$ (resp. $ub(S)$). Finally, we need to consider the bounds of S. We remove v from $ub(S)$ if $|\{i \mid lb(S_i) \subseteq lb(S) \cup \{v\}\}| > max(N)$. Similarly, we add v to $lb(S)$ if $|\{i \mid ub(S_i) \cap ub(S) \setminus \{v\} \neq \emptyset\}| < min(N)$. We can easily extend the soundness proof of Theorem 11 for this algorithm as well. Such an algorithm does not achieve BC (otherwise we would have a polynomial algorithm for a NP-hard problem).

Finally, the constraint Among$([S_1, .., S_n], [T_1, .., T_m], N)$ is very similar to the previous one since several set variables $[T_1, \ldots, T_m]$ behave like their union. That

Algorithm 4. BC for DISJOINT($[S_1, .., S_n], [T_1, .., T_m]$).

1 $S := \bigcup_{i \in \{1,..,n\}} (lb(S_i))$;
2 $T := \bigcup_{i \in \{1,..,m\}} (lb(T_i))$;
3 **if** $S \cap T \neq \emptyset$ **then** fail;
4 **foreach** S_i **do** $ub(S_i) := ub(S_i) \setminus T$;
5 **foreach** T_j **do** $ub(T_j) := ub(T_j) \setminus S$;

is, AMONG($[S_1, .., S_n], [T_1, .., T_m], N$) is similar to AMONG($[S_1, .., S_n], T, N$) with $T = \bigcup_{j \in \{1..m\}} T_j$.

4.2 Common Constraint

We may also want to post a COMMON constraint on set variables. We have:

$$\text{COMMON}(N, M, [S_1, .., S_n], [T_1, .., T_m]) \text{ iff}$$
$$N = |\{i \mid \exists j \ . \ S_i \cap T_j \neq \emptyset\}| \wedge M = |\{j \mid \exists i \ . \ S_i \cap T_j \neq \emptyset\}|$$

Enforcing BC on such a constraint is intractable as it is an extension of the previous AMONG constraint. We can reduce AMONG($[S_1, .., S_n], [T_1, .., T_m], N$) to COMMON($N, M$, $[S_1, .., S_n], [T_1, .., T_m]$) by setting M to $\{0, .., m\}$.

Since we have a means of propagation for AMONG($[S_1, .., S_n], [T_1, .., T_m], N$), we can use it to propagate the COMMON constraint by decomposing it into two such AMONG constraints. However, such a decomposition hurts propagation. Consider the set variables S_1, S_2, S_3, T_1, T_2 and T_3 with $\{i\} \subseteq S_i \subseteq \{i\}$, $N = 1$, $\{\} \subseteq T_i \subseteq \{i\}$, and $M = 2$. The two AMONG constraints of the decomposition are BC while COMMON is inconsistent.

4.3 Disjoint Constraint

We finally consider DISJOINT($[S_1, .., S_n], [T_1, .., T_m]$). Unlike GAC on DISJOINT with integer variables, we can maintain BC on DISJOINT($[S_1, .., S_n], [T_1, .., T_m]$) in polynomial time.

Theorem 13. *Algorithm 4 maintains BC on* DISJOINT($[S_1, .., S_n], [T_1, .., T_m]$) *and runs in* $O((n + m)d)$.

Proof. We first show that the algorithm is sound. If there exists one value which occurs both in the lower bound of one of S_i and in the lower bound of one of the T_j, then we necessarily fail (Line 3). If a value is consumed by one of the T_j, then we cannot satisfy the constraint if this value is allowed to be consumed by one of the S_i (Line 4). Similarly, if a value is consumed by one of the S_i, then we cannot satisfy the constraint if this value is allowed to be consumed by one of the T_j (Line 5).

To show completeness, we prove that either we fail or we return bound consistent domains. We only consider the S_i as the reasoning is analogous for the T_j.

First, if the lower bounds of the S_i do not overlap those of the T_j, then assigning all S_i and T_j to their lower bound is a solution. Thus, the lower bounds are BC. Now, for each value in the upper bound of each S_i, we can construct a satisfying assignment involving v by assigning all other S_i to their lower bounds, S_i to its lower bound plus the element v, and all the T_j to their corresponding lower bounds as none has v as an element.

If d is the total number of values appearing in the upper bounds of the set variables, then at worst case the complexity of line 4 is $O(nd)$ and of line 5 is $O(md)$. Hence, the algorithm runs in $O((n+m)d)$. ♡

Note that if we add a cardinality restriction to the size of the set variables, it becomes NP-hard to enforce BC on this constraint.

5 Related Work

Among$([X_1,..,X_n],[k_1,..,k_m],N)$ was first introduced in CHIP by [3]. A closely related constraint is the Count constraint [11]. Count$([X_1,..,X_n],v,op,N)$ where $op \in \{\leq,\geq,<,>,\neq,=\}$ holds iff N op $|\{i \mid X_i = v\}|$. The Among constraint is more general as it counts the variables taking values from a set whereas Count counts those taking a given value. The algorithm of Among can easily be adapted to cover the operations considered in Count.

There are other counting and related constraints for which there are specialised propagation algorithms such as Gcc [9], NValue [1], Same and UsedBy [4].

In [6], a wide range of counting and occurrence constraints are specified using two primitive global constraints, Roots and Range. For instance, the Among on integer variables constraint is decomposed into a Roots and set cardinality constraint. Similarly, the Common constraint is decomposed into two Roots, two Range and two set cardinality constraints. However, Roots and Range cannot be used to express the Among, Common, and Disjoint constraints on set variables.

Finally our approach to the study of the computational complexity of reasoning with global constraints has been proposed in [5]. In particular, as in [5], we show how computational complexity can be used to determine when a lesser level of local consistency should be enforced and when decomposing constraints will lose pruning.

6 Conclusions

We have studied a number of variations of the Among, Common and Disjoint constraints over integer and set variables. Such constraints are useful in modelling problems involving resources. Our study shows that whether a global constraint is tractable or not can be easily affected by a slight generalization or specialization of the constraint. However, a propagation algorithm for an intractable constraint can often be adapted from a propagation algorithm of a

Table 1. Summary of complexity results

Constraint	Tractability
Integer Variables	
$\textsc{Among}([X_1, .., X_n], \mathcal{K}, N)$	GAC is in P
$\textsc{Among}([X_1, .., X_n], S, N)$	HC is NP-hard
$\textsc{Among}([X_1, .., X_n], [Y_1, .., Y_m], N)$	GAC is NP-hard
$\textsc{Common}(N, M, [X_1, .., X_n], [Y_1, .., Y_m])$	GAC is NP-hard
$\textsc{Disjoint}([X_1, .., X_n], [Y_1, .., Y_m])$	GAC is NP-hard, BC is in P
Set Variables	
$\textsc{Among}([S_1, .., S_n], \mathcal{K}, N)$	BC is in P
$\textsc{Among}([S_1, .., S_n], S, N)$	BC is NP-hard
$\textsc{Among}([S_1, .., S_n], [T_1, .., T_m], N)$	BC is NP-hard
$\textsc{Common}(N, M, [S_1, .., S_n], [T_1, .., T_m])$	BC is NP-hard
$\textsc{Disjoint}([S_1, .., S_n], [T_1, .., T_m])$	BC is in P

similar tractable one. In Table 1, we present a summary of our complexity results. For integer variables, we propose a polynomial time propagation algorithm for the Among constraint that achieves GAC. We prove that Among is intractable when we count the number of variables using values from *a set variable* or a *sequence of integer variables*. Nevertheless, we propose a polynomial algorithm to propagate the former and show how this algorithm can be used to propagate the latter. We also show that the Common constraint is intractable in general, and this holds even in the special case of the Disjoint constraint when the number of values in common is zero. The last result is somewhat surprising, since the Disjoint constraint is related to (and weaker than) the AllDifferent constraint. When we demonstrate the intractability of a constraint like Disjoint, we also present a polynomial method to propagate the constraint. Finally, we consider Among, Common and Disjoint constraints over set variables rather than integer variables. We show that most of the results on integer variables hold for set variables with the exception that the Disjoint constraint now becomes tractable.

In future work, we will focus on determining whether BC on $\textsc{Among}([X_1, .., X_n], S, N)$ is tractable or not. Such a result will also help us answer the still open questions of whether BC on the related $\textsc{Among}([X_1, .., X_n], [Y_1, .., Y_m], N)$ and $\textsc{Common}(N, M, [X_1, .., X_n], [Y_1, .., Y_m])$ is tractable or not. We will also implement these constraints and see their value in practice.

Acknowledgements

Hebrard and Walsh are supported by the National ICT Australia, which is funded through the Australian Government's *Backing Australias Ability* initiative, in part through the Australian Research Council. Hnich received support from Science Foundation Ireland (Grant 00/PI.1/C075). We would like to thank our reviewer for helping to improve the paper.

References

1. Beldiceanu, N. 2001. Pruning for the minimum constraint family and for the number of distinct values constraint family. In *Proc. of CP 2001*, 211–224. Springer.
2. Beldiceanu, N. 2000. Global constraints as graph properties on a structured network of elementary constraints of the same type. Technical report T2000/01, Swedish Institute of Computer Science.
3. Beldiceanu, N., and Contegean, E. 1994. Introducing global constraints in CHIP. *Mathematical Computer Modelling* 20(12):97–123.
4. Beldiceanu, N., Katriel, I., and Thiel, S. 2004. Filtering algorithms for the *same* and *usedby* constraints. MPI Technical Report MPI-I-2004-1-001.
5. Bessiere, C., Hebrard, E., Hnich, B. and Walsh, T. 2004. The Complexity of Global Constraints. In *Proc. of AAAI 2004*. AAAI Press / The MIT Press.
6. Bessiere, C., Hebrard, E., Hnich, B., Kizilitan Z. and Walsh, T. 2005. The Range and Roots Constraints: Specifying Counting and Occurrence Problems. In *Proc. of IJCAI 2005*, 60–65. Professional Book Center.
7. Cheng, B.M.W., Choi, K.M.F., Lee, J.H.M. and Wu, J.C.K. 1999. Increasing Constraint Propagation by Redundant Modeling: an Experience Report. *Constraints* 4(2): 167–192.
8. Régin, J.-C. 1994. A filtering algorithm for constraints of difference in CSPs. In *Proc. of AAAI 1994*, 362–367. AAAI Press.
9. Régin, J.-C. 1996. Generalized arc consistency for global cardinality constraints. In *Proc. of AAAI 1996*, 209–215. AAAI Press / The MIT Press.
10. Sadler, A., and Gervet, C. 2001. Global reasoning on sets. In *Proc. of Workshop on Modelling and Problem Formulation (FORMUL'01)*. Held alongside CP 2001.
11. Swedish Institue of Computer Science. 2004. *SICStus Prolog User's Manual, Release 3.12.0*. Available at `http://www.sics.se/sicstus/docs/latest/pdf/sicstus.pdf`.