Propagating Logical Combinations of Constraints

Fahiem Bacchus[*]
University of Toronto
Toronto, Canada
fbacchus@cs.toronto.edu

Toby Walsh[†]
NICTA and UNSW
Sydney, Australia
tw@cse.unsw.edu.au

Abstract

Many constraint toolkits provide logical connectives like disjunction, negation and implication. These permit complex constraint expressions to be built from primitive constraints. However, the propagation of such complex constraint expressions is typically limited. We therefore present a simple and light weight method for propagating complex constraint expressions. We provide a precise characterization of when this method enforces generalized arc-consistency. In addition, we demonstrate that with our method many different global constraints can be easily implemented.

1 Introduction

Real world problems often contain logical combinations of numerical and symbolic constraints. For example, many configuration problems are naturally specified using implication and other logical connectives (e.g. if the car is a coupe or a convertible then a roof rack is not a valid option). To facilitate the modeling of such problems, constraint toolkits allow logical combinations of primitive constraints to be posted. However, such combinations are typically not propagated very effectively. For example, the propagation of a disjunction of constraints is generally delayed until all but one of the disjuncts are falsified after which the remaining disjunct (which must now hold) is propagated.

We will present a simple and light weight method to propagate constraint expressions built up from logical connectives and primitive constraints. This method can be incorporated into any current constraint toolkit by a simple extension to the propagators for the primitive constraints. We can therefore provide the user with a rich language for specifying problems, whilst preserving the ability to prune the search space. To demonstrate the usefulness of such a facility, we show that many global constraints can be easily specified and, in some cases, effectively propagated, using simple constraint expressions. Thus our method can often provide the toolkit user with a very low cost alternative to the enterprise of designing and implementing special purpose propagators for unusual global constraints that might appear in their problem.

[*]This research funded by NSERC Canada.
[†]This research funded in part by National ICT Australia.

2 Inconsistent and Valid Assignments

A *constraint satisfaction problem* consists of a set of variables, each with a domain of values, and a set of constraints. Each constraint consists of a scope of variables to which it is applied, and a relation of allowed values for those variables. For convenience, we represent the domains $\mathcal{D}$ of the variables by the set of possible assignments. For example, if we have two 0/1 variables, X and Y then $\mathcal{D} = \{X = 0, X = 1, Y = 0, Y = 1\}$. We let $domain(X)$ be the set of values in the domain of the variable X: $domain(X) = \{a \mid X = a \in \mathcal{D}\}$. An **assignment set** τ is a set of assignments to variables such that no variable is assigned more than one value by τ. The **scope** of an assignment set τ (constraint C) is the set of variables in τ (C) and is denoted by $scope(\tau)$ ($scope(C)$). Given a constraint C and an assignment set τ with $scope(C) \subseteq scope(\tau)$, we write $C(\tau)$ iff the assignments in τ satisfies C. That is $C(\tau)$ iff there exists $X_1 = a_1, .., X_k = a_k \in \tau$ with $scope(C) = \{X_1, .., X_k\}$ and $X_1 = a_1, .., X_k = a_k$ satisfies C. We write $\neg C(\tau)$ otherwise.

An assignment is (generalized arc) **inconsistent** for a constraint iff no assignment set containing it satisfies the constraint. That is, $X = a$ is inconsistent for C iff $\forall \tau.\big(scope(C) \subseteq scope(\tau) \land X = a \in \tau\big) \rightarrow \neg C(\tau)$. A constraint C has an unique maximal set of inconsistent assignments $MaxInc(C)$. For example, given the constraint $X < Y$ with $X = \{0, 1, 2\}$ and $Y = \{1, 2\}$, then $\{X = 2\}$ is the maximal set of inconsistent assignments. Assignments that are **consistent** have at least one witness falsifying the above condition; i.e., $X = a$ is consistent iff there is an assignment set τ (called a **support**) with $scope(\tau) = scope(C) \land X = a \in \tau \land C(\tau)$. A constraint C is **GAC** (Generalized Arc Consistent) iff every value of every variable in $scope(C)$ has at least one support. If $X = a$ is inconsistent, we can prune a from the domain of X. A constraint C can be made GAC by simply pruning all values in $MaxInc(C)$ from the domains of their respective variables.

An essential notion for our approach is the concept of a **valid assignment**. Valid assignments are the dual of inconsistent assignments. An assignment is **valid** for a constraint iff all assignment sets containing it satisfy the constraint. That is, $X = a$ is valid for C iff $\forall \tau.\big(scope(C) \subseteq scope(\tau) \land X = a \in \tau\big) \rightarrow C(\tau)$. As with inconsistent values every constraint C has a unique maximal set of valid assignments, $MaxValid(C)$. For example, given the constraint $X < Y$

with $X = \{0, 1, 2\}$ and $Y = \{1, 2\}$, the maximal set of valid assignments is $\{X = 0\}$. All possible extensions of $X = 0$ satisfy the constraint $X < Y$, but all other assignments to X can be extended so that they fail to satisfy the constraint.

All of the concepts presented for inconsistent assignments have dual versions for valid assignments. For example, the dual of consistent assignments is the notion of non-valid assignments. An assignment $X = a$ is **non-valid** if there is at least one assignment set τ with $scope(\tau) = scope(C) \wedge X = a \in \tau \wedge \neg C(\tau)$. Another example is that the dual of a support is a non-support. An assignment set τ is a **non-support** for an assignment $X = a$ in a constraint C iff $scope(\tau) = scope(C) \wedge X = a \in \tau \wedge \neg C(\tau)$. A non-support witnesses the non-validity of $X = a$.

3 Constraint Expressions

To build complex constraints, we combine primitive constraints using negation, disjunction and conjunction. A **constraint expression** is either a primitive constraint C or any well-founded Boolean expression of the form: *true*, *false*, $not(\mathcal{C}_1)$, $or(\mathcal{C}_1, .., \mathcal{C}_k)$ or $and(\mathcal{C}_1, .., \mathcal{C}_k)$, where each $\mathcal{C}_i$ is itself a constraint expression. *true* is the primitive constraint which is always valid, whilst *false* is always inconsistent. We also allow the expressions $implies(\mathcal{C}_1, \mathcal{C}_2)$, $iff(\mathcal{C}_1, \mathcal{C}_2)$, $xor(\mathcal{C}_1, \mathcal{C}_2)$ and $ifthen(\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3)$, but regard these additional connectives to be abbreviations:

$$\begin{aligned}
implies(\mathcal{C}_1, \mathcal{C}_2) &\leftrightarrow or(not(\mathcal{C}_1), \mathcal{C}_2) \\
iff(\mathcal{C}_1, \mathcal{C}_2) &\leftrightarrow and(or(not(\mathcal{C}_1), \mathcal{C}_2), or(not(\mathcal{C}_2), \mathcal{C}_1) \\
xor(\mathcal{C}_1, \mathcal{C}_2) &\leftrightarrow and(or(\mathcal{C}_1, \mathcal{C}_2), or(not(\mathcal{C}_1), not(\mathcal{C}_2)) \\
ifthen(\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3) &\leftrightarrow and(or(not(\mathcal{C}_1), \mathcal{C}_2), or(\mathcal{C}_1, \mathcal{C}_3))
\end{aligned}$$

Each constraint expression $\mathcal{C}$ represents a new constraint whose scope is equal to the union of the scopes of the primitive constraints in $\mathcal{C}$. An assignment set τ satisfies $\mathcal{C}$ iff $scope(\mathcal{C}) \subseteq scope(\tau)$ and the Boolean expression representing $\mathcal{C}$ evaluates to true given the truth values of the component primitive constraints under τ. For example, an absolute value constraint $X = abs(Y)$ can be written as the constraint expression $ifthen(Y \geq 0, X = Y, X = -Y)$. Similarly, a max constraint $X = max(Y, Z)$ can be written as the constraint expression $and(X \geq Y, X \geq Z, or(X = Y, X = Z))$.

Since a constraint expression is itself a constraint, associated with every constraint expression $\mathcal{C}$ is a maximal set of inconsistent assignments, $MaxInc(\mathcal{C})$, and a maximal set of valid assignments $MaxValid(\mathcal{C})$. We can make the constraint expression $\mathcal{C}$ generalized arc consistent by pruning all assignments in $MaxInc(\mathcal{C})$. It is also useful to observe that the duality between valid assignments and inconsistent assignments implies $MaxInc(\mathcal{C}) = MaxValid(not(\mathcal{C}))$: $X{=}a \in MaxInc(\mathcal{C}) \leftrightarrow (\forall \tau . X{=}a \in tau \rightarrow (\neg\mathcal{C}(\tau) \leftrightarrow not(\mathcal{C})(\tau))) \leftrightarrow X{=}a \in MaxValid(not(\mathcal{C}))$.

4 Constraint Propagation

Not surprisingly, it is NP-hard to compute $MaxInc$ for an arbitrary constraint expression. More precisely, deciding if a set of assignments is *the maximal* inconsistent set for an arbitrary constraint expression is D^P-complete in general.[1] There are results on tractable languages that identify tractable cases for computing $MaxInc$ and $MaxValid$. However, we are interested here in computing inconsistent assignments for arbitrary constraint expressions, which is intractable in general.

$Inc(not(\mathcal{C}_1), \mathcal{D})$	$=$	$Valid(\mathcal{C}_1, \mathcal{D})$
$Valid(not(\mathcal{C}_1), \mathcal{D})$	$=$	$Inc(\mathcal{C}_1, \mathcal{D})$
$Inc(or(\mathcal{C}_1, .., \mathcal{C}_k), \mathcal{D})$	$=$	$\bigcap_i Inc(\mathcal{C}_i, \mathcal{D})$
$Valid(and(\mathcal{C}_1, .., \mathcal{C}_k), \mathcal{D})$	$=$	$\bigcap_i Valid(\mathcal{C}_i, \mathcal{D})$
$Inc(and(\mathcal{C}_1, .., \mathcal{C}_k), \mathcal{D})$	$=$	$ItInc(and(\mathcal{C}_1, .., \mathcal{C}_k), \mathcal{D})$
$Valid(or(\mathcal{C}_1, .., \mathcal{C}_k), \mathcal{D})$	$=$	$ItValid(or(\mathcal{C}_1, .., \mathcal{C}_k), \mathcal{D})$

$ItInc(and(\mathcal{C}_1, .., \mathcal{C}_k), \mathcal{D})$	$ItValid(or(\mathcal{C}_1, .., \mathcal{C}_k), \mathcal{D})$
inc := $\emptyset$	valid := $\emptyset$
repeat	**repeat**
inc$'$:= $\bigcup_i Inc(\mathcal{C}_i, \mathcal{D})$	valid$'$:= $\bigcup_i Valid(\mathcal{C}_i, \mathcal{D})$
$\mathcal{D}$:= $\mathcal{D} -$ inc$'$	$\mathcal{D}$:= $\mathcal{D} -$ valid$'$
inc := inc $\cup$ inc$'$	valid := valid $\cup$ valid$'$
until(inc$' = \emptyset$)	**until**(valid$' = \emptyset$)
return(inc)	**return**(valid)

Table 1: Functions for computing valid and inconsistent assignments of a constraint expression. In addition, $Inc(true, \mathcal{D}) = Valid(false, \mathcal{D}) = \emptyset$, and $Valid(true, \mathcal{D}) = Inc(false, \mathcal{D}) = \mathcal{D}$.

We propose a simple and light weight method for computing subsets of $MaxInc$ and $MaxValid$ in polynomial time. The method is compositional, computing the inconsistent and valid assignments of a constraint expression from the inconsistent and valid assignments of its parts. For a constraint expression $\mathcal{C}$ and variable domains $\mathcal{D}$, the functions $Inc(\mathcal{C}, \mathcal{D})$ and $Valid(\mathcal{C}, \mathcal{D})$ return subsets of $MaxInc(\mathcal{C})$ and $MaxValid(\mathcal{C})$ respectively. These functions recursively apply the rules in Table 1, until they reach the primitive constraints. We assume that each primitive constraint has a poly-time algorithm to compute inconsistent and valid assignments.

The algorithm can be optimized by a simple caching scheme in which we remember the previously computed value $Inc(\mathcal{C}_i, \mathcal{D})$ for each subexpression $\mathcal{C}_i$. If in a subsequent call $Inc(\mathcal{C}_i, \mathcal{D}')$, $\mathcal{D}'$ is identical to $\mathcal{D}$ when restricted to the variables in $scope(\mathcal{C}_i)$, then we can reuse the previously computed result for $Inc(\mathcal{C}_i, \mathcal{D})$. A similar optimization works for $Valid(\mathcal{C}_i, \mathcal{D})$. In addition, if we compute and prune inconsistent values incrementally, we can stop as soon as any variable has a domain wipeout.

4.1 Entailment and Disentailment

A constraint expression is **entailed** iff it holds for all possible assignments. A constraint is **disentailed** iff it does not hold for any possible assignment. As we show below, $Valid(\mathcal{C}, \mathcal{D})$ returns only valid assignments. Hence, if $Valid(\mathcal{C}, \mathcal{D})$ equals the domains of all of the variables in the scope of $\mathcal{C}$, $\mathcal{C}$ must be entailed. In such a situation, we modify the computation of $Valid$ so that $Valid(\mathcal{C}, \mathcal{D}) = \mathcal{D}$ (note that $\mathcal{D}$ might include domains of other variables besides those in the $scope(\mathcal{C})$). Similarly, if $Inc(\mathcal{C}, \mathcal{D})$ equals the domains of all of the vari-

[1] The complexity class D^P contains problems which are the conjunction of a problem in NP and one in coNP [Papadimitriou and Yannakakis, 1984].

ables in the scope of $\mathcal{C}$ then $\mathcal{C}$ is disentailed; and we modify the computation of *Inc* so that $Inc(\mathcal{C}, \mathcal{D}) = \mathcal{D}$.

To show the benefit, consider $implies(even(X), odd(Y))$ with $\mathcal{D}$=$\{X$=0, X=2, Y=1, Y=2$\}$. Now $Inc(implies(even(X), odd(Y)), \mathcal{D}) = Valid(even(X), \mathcal{D}) \cap Inc(odd(Y), \mathcal{D})$. Using the unmodified versions, $Valid(even(X), \mathcal{D})$ will return just valid values for X, whilst $Inc(odd(Y)), \mathcal{D})$ returns just inconsistent values for Y. *Inc* would then compute the empty set of inconsistent assignments for $implies(even(X), odd(Y))$. Note, however, that $domain(X)$ only contains even numbers. Hence $even(X)$ is entailed. Therefore the modified $Valid(even(X), \mathcal{D})$ can return $\mathcal{D}$, in which case $Inc(implies(even(X), odd(Y), \mathcal{D}) = \mathcal{D} \cap Inc(odd(Y), \mathcal{D}) = \{Y{=}2\}$. As required, this is the maximal set of inconsistent assignments.

4.2 Correctness

We prove that *Inc* and *Valid* are correct. That is, they only return inconsistent and valid assignments respectively.

Theorem 1 *Inc and Valid are correct assuming that inconsistent and valid assignments are correctly computed for the constituent primitive constraints.*

Proof: By induction on the structure of the constraint expression. The base case holds by assumption. The step case uses case analysis.

For a constraint expression $not(\mathcal{C}_1, \mathcal{D})$, we have that $Inc(not(\mathcal{C}_1), \mathcal{D}) = Valid(\mathcal{C}_1, \mathcal{D})$. By induction, the assignments in $Valid(\mathcal{C}_1, \mathcal{D})$ are valid. Hence all of these assignments are inconsistent for $not(\mathcal{C}_1, \mathcal{D})$. A dual argument shows that the assignments in $Valid(not(\mathcal{C}_1), \mathcal{D})$ are valid.

For a constraint expression $\mathcal{C} = and(\mathcal{C}_1, .., \mathcal{C}_k)$, we have $Valid(\mathcal{C}, \mathcal{D}) = \bigcap_i Valid(\mathcal{C}_i, \mathcal{D})$. Suppose $X = a \in Valid(\mathcal{C}, \mathcal{D})$. Then for all i, $X = a \in Valid(\mathcal{C}_i, \mathcal{D})$. By the induction hypothesis, the assignments in each $Valid(\mathcal{C}_i)$ are valid. Consider any assignment set τ such that $X = a \in \tau$ and $scope(\mathcal{C}) \subseteq scope(\tau)$. Since $X = a$ is valid for each $\mathcal{C}_i$, τ must satisfy all $\mathcal{C}_i$ and thus must satisfy the conjunction. Hence $X = a$ is also valid for $\mathcal{C}$. For $\mathcal{C} = Inc(and(\mathcal{C}_1, .., \mathcal{C}_k), \mathcal{D})$ a similar argument shows that the assignments in $\bigcup_i Inc(\mathcal{C}_i, \mathcal{D})$ are inconsistent. Deleting these assignments from $\mathcal{D}$ cannot cause any consistent assignment to lose its support, hence $Inc(\mathcal{C}_i, \mathcal{D}')$ on the reduced domain $\mathcal{D}'$ must still return inconsistent assignments. *ItInc* then recomputes $\bigcup_i Inc(\mathcal{C}_i, \mathcal{D})$ until we reach a fixed point.

Similar arguments hold for constraint expressions of the form $or(\mathcal{C}_1, .., \mathcal{C}_k)$. $\Box$

4.3 Termination

The *ItInc* and *ItValid* functions only require a linear (in the size of the CSP) number of iterations to reach their fixed point in the worst case.

Theorem 2 *ItInc and ItValid take $O(nd)$ iterations to reach their fixed points for a constraint expression with n variables and domains of size d. There exist constraint expressions which take $\Theta(nd)$ iterations to reach the fixed point.*

Proof: As each iteration removes at least one value, we must reach the fixed point in at most nd steps. We can give a simple example in which this bound is reached. Consider $and(\mathcal{C}_1, .., \mathcal{C}_n)$ where $\mathcal{C}_i$ is $X_i = X_{i+1}$ for $i < n$ and $X_1 - X_n = 1$ for $i = n$. Suppose $domain(X_i) = \{1, .., d\}$ for every i. Then in the first iteration, *ItInc* returns $\{X_1 = 1\}$ as this value is not supported in $X_1 - X_n = 1$. After this is pruned from $\mathcal{D}$, a second iteration returns $\{X_2 = 1\}$ as this value is now not supported in $X_1 = X_2$. And so on up to the nth iteration which returns $\{X_n = 1\}$. After this is pruned, the $n + 1$th iteration returns $\{X_1 = 2\}$ as this value is now not supported in $X_1 - X_n = 1$. Hence, there are nd iterations before all the values of all the variables are removed. Note that even if we stop when the first variable has a domain wipeout, it will still take $(n-1)d + 1$ iterations. $\Box$

4.4 Maximality

These functions do not always compute maximal sets, even if maximal sets are computed for the primitive constraints from which they are composed. This is not surprising given that computing maximal sets for an arbitrary constraint expression is intractable in general. The following result precisely characterizes when *Inc* returns the maximal inconsistent set of assignments. In other words, the following result identifies exactly when pruning the values returned by *Inc* ensures that a constraint expression is GAC.

We start with a number of definitions. A *hypergraph* $\mathcal{H} = (H, E^H)$ is a set of vertices H and hyperedges E^H each of which is a subset of H. A hypergraph $\mathcal{H}$ has an **acyclic tree decomposition** [Flum *et al.*, 2002] iff there is a tree T satisfying: (1) there is a one-to-one correspondence between the hyperedges of $\mathcal{H}$ and the nodes of T; the hyperedge corresponding to a tree node t is called t's label ($label(t)$); (2) for every vertex $v \in \mathcal{H}$ the set of nodes t of T such that $v \in label(t)$ form a subtree of T. The hypergraph of a conjunctive or disjunctive constraint expression, $\mathcal{C} = and(\mathcal{C}_1, .., \mathcal{C}_k)$ or $\mathcal{C} = or(\mathcal{C}_1, .., \mathcal{C}_k)$, has the variables in $scope(\mathcal{C})$ as vertices and the sets of variables $scope(\mathcal{C}_i)$, $i = 1, .., k$ as hyperedges. We will relax this definition to take account of (dis)entailment. If $\mathcal{C} = and(\mathcal{C}_1, .., \mathcal{C}_k)$ then we ignore any entailed subexpression when constructing the hypergraph. Similarly, if $\mathcal{C} = or(\mathcal{C}_1, .., \mathcal{C}_k)$ then we ignore any disentailed subexpression. Under this relaxation we define a conjunctive or disjunctive constraint expression to be **acyclic** if its corresponding hypergraph has an acyclic tree decomposition. For example, a conjunction in which the primitive constraints are in a chain, and each has only one variable in common with the previous and next constraint is acyclic. Acyclicity is, however, more general than being a chain. We will use acyclicity to characterize when *Inc* computes *MaxInc*.

Theorem 3 *For any constraint expression $\mathcal{C}$ and any variable domains $\mathcal{D}$, $Inc(\mathcal{C}, \mathcal{D}) = MaxInc(\mathcal{C}, \mathcal{D})$ if:*

1. *$\mathcal{C}$ is a primitive constraint and $Inc(\mathcal{C}, \mathcal{D}) = MaxInc(\mathcal{C}, \mathcal{D})$;*
2. *$\mathcal{C} = not(\mathcal{C}_1)$ and $Valid(\mathcal{C}_1, \mathcal{D}) = MaxValid(\mathcal{C}_1, \mathcal{D})$;*
3. *$\mathcal{C} = or(\mathcal{C}_1, .., \mathcal{C}_k)$ and $Inc(\mathcal{C}_i, \mathcal{D}) = MaxInc(\mathcal{C}_i, \mathcal{D})$ for $i \in (1, .., k)$;*
4. *$\mathcal{C} = and(\mathcal{C}_1, .., \mathcal{C}_k)$ and (a) $Inc(\mathcal{C}_i, \mathcal{D}) = MaxInc(\mathcal{C}_i, \mathcal{D})$ for $i \in (1, .., k)$; (b) $\mathcal{C}$ is acyclic; and (c) $|scope(\mathcal{C}_i) \cap scope(\mathcal{C}_j)| \leq 1$ for $i, j \in (1, .., k)$;*

5. $Inc(C, \mathcal{D}) = \mathcal{D}$.

Proof: 1. Immediate.

2. Suppose $Valid(\mathcal{C}_1, \mathcal{D})$ is maximal. Then for any $X = a \notin Valid(\mathcal{C}_1, \mathcal{D})$, there exists τ with $X = a \in \tau$ and $\neg\mathcal{C}_1(\tau)$. Hence $X = a$ cannot be in $Inc(not(\mathcal{C}_1), \mathcal{D})$ as τ is one assignment that prevents it being inconsistent. Hence $Inc(not(\mathcal{C}_1), \mathcal{D})$ is maximal.

3. Suppose $Inc(\mathcal{C}_i, \mathcal{D})$ are maximal. Consider $X = a \notin Inc(or(\mathcal{C}_1, .., \mathcal{C}_k), \mathcal{D})$. Then $X = a \notin \bigcap_{1 \leq i \leq k} Inc(\mathcal{C}_i, \mathcal{D})$. That is, $X = a \notin Inc(\mathcal{C}_j, \mathcal{D})$ for some $j \in (1, .., k)$. As $Inc(\mathcal{C}_j, \mathcal{D})$ is maximal, there exists τ with $X = a \in \tau$ and $\mathcal{C}_j(\tau)$. Thus $X = a$ cannot be in $Inc(or(\mathcal{C}_1, .., \mathcal{C}_k), \mathcal{D})$ as τ is one assignment that prevents it being inconsistent. Hence $Inc(or(\mathcal{C}_1, .., \mathcal{C}_k), \mathcal{D})$ is maximal.

4. Suppose $X \in scope(\mathcal{C})$ with $X = a \in \mathcal{D}$, but $X = a \notin Inc(\mathcal{C}, \mathcal{D})$. We must show that $X = a \notin MaxInc(\mathcal{C}, \mathcal{D})$. Let $\mathcal{D}^c = \mathcal{D} - Inc(\mathcal{C}, \mathcal{D})$, i.e., the consistent assignments remaining in the variable domains. From Table 1 we observe that $Inc(\mathcal{C}_i, \mathcal{D}^c) = \emptyset$ and by condition (a) $MaxInc(\mathcal{C}_i, \mathcal{D}^c) = \emptyset$ for all $i \in (1, .., k)$. Consider the acyclic tree decomposition associated with $\mathcal{C}$. Orient this tree so that the root is labeled with $scope(\mathcal{C}_i)$ for some $\mathcal{C}_i$ with $X \in scope(\mathcal{C}_i)$. Note that by property (2) of an acyclic tree decomposition and condition (c), each of the subtrees below $\mathcal{C}_i$ can have at most one variable in common with the other subtrees. Furthermore if two subtrees do have a variable in common that variable must be in the $scope(\mathcal{C}_i)$.

Since $X = a \in \mathcal{D}^c$ it must have some support τ on $\mathcal{C}_i$ such that $\tau \in \mathcal{D}^c$. Now we extend this support downwards in the tree decomposition to the children of $\mathcal{C}_i$: $\mathcal{C}_j^1, .., \mathcal{C}_j^\ell$. Each such child $\mathcal{C}_j$ shares only one variable with $\mathcal{C}_i$, say Y, and Y must be assigned some value in τ, say $Y = b$. Since $Y = b \in \mathcal{D}^c$ it must have a support τ_j in $\mathcal{C}_j$ such that $\tau_j \in \mathcal{D}^c$. Thus we can extend τ to a support for $and(\mathcal{C}_i, \mathcal{C}_j)$ for each child of $\mathcal{C}_i$. Furthermore the supports τ_j for the individual children of $\mathcal{C}_i$ cannot be in conflict: $\mathcal{C}_j$ and $\mathcal{C}_{j'}$ can only share a variable already assigned by τ, hence τ_j and $\tau_{j'}$ must agree with τ and with each other on the value assigned to this variable. Thus we can extend τ to a support for all of $\mathcal{C}_i$'s children. Furthermore, by the same argument each support τ_j for the child $\mathcal{C}_j$ can be extended to a support for all of the conjuncts in the subtree below $\mathcal{C}_j$. Hence, τ can be extended to a support for all of $\mathcal{C}$, and since $X = a \in \tau$, $X = a \notin MaxInc(\mathcal{C}, \mathcal{D})$.

Note that if $and(\mathcal{C}_1, .., \mathcal{C}_k)$ contains any entailed conjuncts, these can be eliminated without changing the maximal set of inconsistent assignments. We can then apply the argument above to the remaining acyclic part of the conjunction.

5. Immediate since $MaxInc(\mathcal{C}, \mathcal{D}) \subseteq \mathcal{D}$. □

In fact, we can show that these five cases are the only ones in which *Inc* is always guaranteed to be maximal. This reverse direction needs a little care as *Inc* may compute *MaxInc* by chance. However, these five cases are the only ones in which, irrespective of the constraint subexpressions, *Inc* is guaranteed to compute *MaxInc*. A dual result holds, and characterizes precisely when *Valid* computes *MaxValid*.

Previous related results have shown that acyclic database queries of bounded tree width are tractable. Bounded tree width (tree width one) is also known to be the condition that characterizes when arc consistency achieves GAC on conjunctions of binary constraints. The main difference between our result and these previous results is that we do not place any restriction on the arity of the primitive constraints. Hence, our result does not depend on bounded tree width (the tree width of the constraint graph is at least as large as the arity of the primitive constraints). Given primitive constraints for which GAC can be efficiently computed (no matter their arity), our result characterizes when this efficiency can be lifted to complex combinations of these constraints.

5 Some Applications

To demonstrate the usefulness of constraint expressions and our propagation algorithm, we show that a wide range of global and other constraints can be specified and propagated using simple constraint expressions. In many cases, our light weight propagation algorithm is able to achieve GAC. Our method is then an effective and low cost means to implement these constraints. This is especially valuable when the constraints useful for a particular problem are too specialized to be in the user's constraint toolkit.

DOMAIN constraint. This channels between a variable and a sequence of 0/1 variables representing the possible values taken by the variable. More precisely, DOMAIN$(X, [X_1, .., X_n])$ ensures $X = i$ iff $X_i = 1$ [Refalo, 2000]. We can decompose this into a disjunction:

$$or(and(X = 1, X_1 = 1, .., X_n = 0), .., \\ and(X = n, X_1 = 0, .., X_n = 1))$$

The equality constraints in each conjunct again have no variables in common. Hence pruning the values returned by *Inc* enforces GAC.

ELEMENT constraint. This indexes into an array with a variable [Van Hentenryck and Carillon, 1988]. More precisely, ELEMENT$(I, [X_1, .., X_n],\ J)$ holds iff $X_I = J$. We can use this to look up the price of a particular component included in a configuration problem. It decomposes into a simple disjunction:

$$or(and(I = 1, J = X_1), .., and(I = n, J = X_n))$$

The equality constraints within each conjunct share no variables. Hence pruning the values returned by *Inc* enforces GAC.

MEMBER constraint. This ensures that a particular value is used. More precisely, MEMBER$(I, [X_1, .., X_n])$ holds iff there exists j with $X_j = I$. We can decompose this into a disjunction:

$$or(X_1 = I, .., X_n = I)$$

Pruning the values returned by *Inc* enforces GAC.

MAX constraint. This computes the maximum value taken by a sequence of variables. More precisely, MAX$(N, [X_1, .., X_n])$ holds iff there exists i with $N = X_i$ and $X_i \geq X_j$ for all j. A MAX constraint can be decomposed into a disjunction:

$$or(and(N = X_1, X_1 \geq X_2, .., X_1 \geq X_n) \\ and(N = X_2, X_2 \geq X_1, X_2 \geq X_3, .., X_2 \geq X_n) \\ .., and(N = X_n, X_n \geq X_1, .., X_n \geq X_{n-1}))$$

Pruning the values returned by Inc on such a decomposition enforces GAC.

CHANNELLING constraints. It is often advantageous to specify multiple viewpoints of a problem. Constraints may be easier to specify in one viewpoint and propagate better in another. Channelling constraints are then needed to link the different viewpoints. For example, multiple viewpoints have proved useful in the orchestra rehearsal problem, `prob039` in CSPLIB. The channelling constraints used here can be specified as constraint expressions:

$$iff(X_i = 1, or(X_{i-1} = 1, Y_i = 1))$$
$$iff(Z_k = j, and(X_j = 1, X_{j-1} = 0))$$

The problem also contained a specialized optimality constraint which can be specified as the constraint expression:

$$implies(X < Y, W_X = 1)$$

Where $W_X = 1$ can itself be specified with an ELEMENT constraint.

LEX constraint. This lexicographically orders two vectors of variables. It has many applications when dealing with symmetry [Frisch *et al.*, 2002]. We can decompose it into a disjunction:

$$\begin{aligned}&or(X_1 < Y_1,\\ &\quad and(X_1 = Y_1, X_2 < Y_2),\\ &\quad and(X_1 = Y_1, X_2 = Y_2, X_3 < Y_3), ..,\\ &\quad and(X_1 = Y_1, .., X_{n-1} = Y_{n-1}, X_n \leq Y_n))\end{aligned}$$

As the constraints in each conjunct share no variables, pruning the values returned by Inc enforces GAC on the LEX constraint. In addition, if we memoize previous results, Inc will be comparable to the efficient algorithm in [Frisch *et al.*, 2002].

The choice of decomposition of a global constraint is important for ensuring as much propagation as possible. For instance, an alternative decomposition is the conjunction:

$$\begin{aligned}&and(X_1 \leq Y_1,\\ &\quad implies(X_1 = Y_1, X_2 \leq Y_2), ..\\ &\quad implies(and(X_1 = Y_1, .., X_{n-1} = Y_{n-1}), X_n \leq Y_n))\end{aligned}$$

Pruning the values returned by Inc on this decomposition may not enforce GAC on the LEX constraint. Consider X_1 and $Y_1 \in \{0, 1\}$, $X_2 = 1$ and $Y_2 = 0$. Then $X_1 = 1$ and $Y_1 = 0$ are both inconsistent assignments. However, Inc applied to this decomposition will return the empty set.

VALUE PRECEDENCE constraint. This breaks symmetries in a sequence of variables, X_1 to X_n caused by two indistinguishable values s and t [Law and Lee, 2004]. We can decompose it into a simple conjunction:

$$\begin{aligned}&and(X_1 \neq t, implies(X_2 = t, X_1 = s), ..,\\ &\quad implies(X_n = t, or(X_1 = s, .., X_{n-1} = s)))\end{aligned}$$

Theorem 2 of [Law and Lee, 2004] shows that enforcing GAC on each conjunct achieves GAC on the global constraint. As each conjunct expands out into a simple disjunction, pruning the values returned by Inc enforces GAC on the global constraint.

AMONG constraint. This limits the number of variables taking values from a set [Beldiceanu and Contejean, 1994]. More precisely, AMONG$([X_1, .., X_n], [d_1, .., d_m], N)$ holds iff $N = |\{i | X_i = d_j\}|$. The constraint is useful in many resource allocation and scheduling problems. By introducing additional variables N_i in which to accumulate a count, we can decompose this into a relatively simple constraint expression:

$$\begin{aligned}&and(ifthen(X_1 \in D, N_1 = 1, N_1 = 0),\\ &\quad ifthen(X_2 \in D, N_2 = N_1 + 1, N_2 = N_1),\\ &\quad .., ifthen(X_n \in D, N = N_{n-1} + 1, N = N_{n-1})),\end{aligned}$$

where $X_i \in D$ is an abbreviation for $or(X_i = d_1, .., X_i = d_m)$.

Unfortunately, Inc does not achieve GAC on such a decomposition. Consider, for example, $X_1, X_2 \in \{0, 1\}$, $N \in \{0, 1, 2\}$ and $N_1 \in \{0, 1\}$. Enforcing GAC on AMONG$([X_1, X_2], [0, 1], N)$ will prune 0 and 1 from N whilst Inc returns the empty set. Similar decompositions and results hold for the ATMOST, ATLEAST, CHANGE, and COUNT constraints.

NOTALLEQUAL constraint. This ensures that not all variables take the same value [Beldiceanu and Contejean, 1994]. More precisely, NOTALLEQUAL$([X_1, .., X_n])$ holds iff there exists i and j with $X_i \neq X_j$. This can be decomposed into the following constraint expression:

$$or(X_1 \neq X_2, .., X_1 \neq X_n)$$

Pruning the values returned by Inc enforces GAC.

NVALUE constraint. This counts the number of values used by a sequence of variables [Pachet and Roy, 1999]. More precisely, NVALUE$([X_1, .., X_n], N)$ holds iff $|\{X_i | 1 \leq i \leq n\}| = N$. The constraint is useful in a wide range of problems involving resources The ALLDIFFERENT constraint is a special case of the NVALUE constraint in which $N = n$. Unfortunately, it is NP-hard in general to enforce GAC on a NVALUE constraint [Bessiere *et al.*, 2004].

One way to propagate this constraint in polynomial time is to decompose it with an additional set of 0/1 variables representing the characteristic function of the set of values used:

$$\begin{aligned}&and(iff(S_1 = 1, or(X_1 = 1, .., X_n = 1)),\\ &\quad .., iff(S_m = 1, or(X_1 = m, .., X_n = m)),\\ &\quad \textstyle\sum_{i=1}^{m} S_i = N)\end{aligned}$$

Where the domains of the X_i are assumed to be from 1 to m. Not surprisingly, Inc does not compute the maximal set of inconsistent assignments for this constraint expression. For example, suppose $X_1, X_2 \in \{1, 2\}$, $X_3 \in \{1, 2, 3\}$, $N = 3$ and $S_i = 1$ for $1 \leq i \leq 3$. Then Inc returns the empty set, even though $X_3 = 1$ and $X_3 = 2$ are inconsistent.

A similar decomposition and result holds for the COMMON constraint, which is also NP-hard to propagate.

6 Related Work

Lhomme has proposed GAC algorithms for logical combinations of primitive constraints, focused primarily on those that are given in extension [Lhomme, 2004]. For example, on a disjunction like $or(C_1, .., C_k)$, the algorithm tests each assignment for membership in $\bigcap_i MaxInc(C_i)$. If the assignment is not in $MaxInc(C_i)$ for some i, the remaining $MaxInc$ sets do not have to be tested. However, in the worst case, we still have to compute $MaxInc$ for all of the primitive constraints, as we require in our method.

Bessière and Régin have proposed an algorithm for enforcing GAC on a conjunction of primitive constraints [Bessière and Régin, 1998]. However, as could be expected, this requires $O(d^n)$ time in the worst case.

Other approaches to propagating constraint expressions are based on reification or delaying. It is not hard, however, to show that Inc provides strictly more pruning. Suppose we have a disjunction like $or(X < 2, X > 4)$. We can reify this into $B_1 \leftrightarrow X < 2$, $B_2 \leftrightarrow X > 4$, and $B_1 + B_2 > 0$. These constraints are delayed until either inequality is entailed or disentailed. However, Inc can prune values immediately. Consider $X \in \{1, 3, 5\}$. Then neither inequality is entailed or disentailed. However, Inc will return the inconsistent assignment $X = 3$.

To perform more pruning on such disjunctions, `cc(FD)` introduced constructive disjunction [Van Hentenryck *et al.*, 1998]. If any of the disjuncts, C_i in a constructive disjunction, $C_1 \vee_c .. \vee_c C_k$ is entailed by the constraint store, then the constructive disjunction is satisfied. Otherwise, each constraint C_i is added in turn to the constraint store and propagated. The resulting inconsistent assignments are recorded, the state restored, and the next constraint is processed. The intersection of the inconsistent assignments found for each constraint are then taken to be the inconsistent assignments for the disjunction. Constructive disjunction can do more pruning than enforcing GAC. For example, even though both $or(X = 0, Y = 0)$ and $X = Y$ on 0/1 variable are GAC, propagating $X = 0 \vee_c Y = 0$ and $X = Y$ prunes $X = Y = 1$. This extra pruning arises from interaction between the disjunctive constraint and the other constraints in the constraint store. For this reason, constructive disjunction can be very expensive, and may not justify its costs in practice [J. Würtz and T. Müller, 1996].

The cardinality constraint can be used to implement conjunction, disjunction, negation, as well as a host of other useful constraints [Van Hentenryck and Deville, 1991]. However, only a very restricted form of consistency is enforced on the cardinality constraint, and Inc does more pruning in general. For example, if C_1 is $X = 0$, C_2 is $X = 1$ and $\mathcal{D} = \{X = 0, X = 1, X = 2\}$ then $Inc(or(C_1, C_2), \mathcal{D}) = \{X = 2\}$. Pruning $X = 2$ makes the problem GAC. However, the equivalent cardinality constraint, $card(N, [C_1, C_2])$ where $N \geq 1$ is consistent without any prunings.

In contrast to these previous works, we have provided a tractable method for computing a subset of $MaxInc$ that does a useful amount of constraint propagation. Our algorithm is compositional as it uses the propagators provided for the primitive constraints. Hence it can be applied to complex, *nested* logical expressions.

7 Conclusion

We have proposed a simple and light weight method for propagating logical combinations of primitive constraints. Since computing the maximal set of inconsistent assignments for such constraint expressions is intractable in general, we have given a polynomial time function which computes a tractable subset compositionally. We characterized precisely when this function computes the maximal set of inconsistent assignments. Finally we have shown that many different global constraints can be implemented using these methods. There remain many interesting directions to follow both from a theoretical and practical perspective. For example, how do we compute and use nogoods for constraint expressions?

References

[Beldiceanu and Contejean, 1994] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20:97–123, no. 12 1994.

[Bessière and Régin, 1998] C. Bessière and J-C. Régin. Local consistency on conjunctions of constraints. In *Proc, of ECAI-98 Workshop on Non-Binary Constraints*. 1998.

[Bessiere *et al.*, 2004] C. Bessiere, E. Hebrard, B. Hnich, and T. Walsh. The complexity of global constraints. In *Proc. of the 19th National Conf. on AI*. AAAI, 2004.

[Flum *et al.*, 2002] Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree-decompositions. *J. ACM*, 49(6):716–752, 2002.

[Frisch *et al.*, 2002] A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In *8th Int. Conf. on Principles and Practices of Constraint Programming (CP-2002)*. Springer, 2002.

[Law and Lee, 2004] Y.C. Law and J.H.M. Lee. Global constraints for integer and set value precedence. In *8th Int. Conf. on Principles and Practice of Constraint Programming (CP2004)*, pages 362–376. Springer, 2004.

[Lhomme, 2004] O. Lhomme. Arc-consistency Filtering Algorithms for Logical Combinations of Constraints. In *Proc. of Int. Conf. on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems* (CP-AI-OR'04). Springer, 2004.

[Pachet and Roy, 1999] F. Pachet and P. Roy. Automatic generation of music programs. In J. Jaffar, editor, *Proc. of 5th Int. Conf. on Principles and Practice of Constraint Programming (CP99)*, pages 331–345. Springer, 1999.

[Papadimitriou and Yannakakis, 1984] C. Papadimitriou and M. Yannakakis. The complexity of facets (and some facets of complexity). *Journal of Computer and System Sciences*, 28(2):244–259, 1984.

[Refalo, 2000] P. Refalo. Linear formulation of constraint programming models and hybrid solvers. In R. Dechter, editor, *Proc. of 6th Int. Conf. on Principles and Practice of Constraint Programming (CP2000)*, pages 369–383. Springer, 2000.

[Van Hentenryck and Carillon, 1988] P. Van Hentenryck and J.-P. Carillon. Generality versus specificity: An experience with AI and OR techniques. In *Proc. of 7th National Conf. on Artificial Intelligence*, pages 660–664. AAAI, 1988.

[Van Hentenryck and Deville, 1991] P. Van Hentenryck and Y. Deville. The cardinality operator: a new logical connective for constraint logic programming. In *Proc. of the Int. Conf. on Logic Programming (ICLP 91)*, pages 745–759, 1991.

[Van Hentenryck *et al.*, 1998] P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation and evaluation of the constraint language cc(fd). *Journal of Logic Programming*, 37(1–3):139–164, 1998.

[J. Würtz and T. Müller, 1996] J. Würtz and T. Müller. Constructive disjunction revisited. In *Proc. of 20th German Annual Conf. on Artificial Intelligence*. Springer-Verlag, 1996.