The Complexity of Reasoning with Global Constraints

Christian Bessiere
LIRMM, CNRS/U. Montpellier
Montpellier, France
bessiere@lirmm.fr

Emmanuel Hebrard
4C and UCC
Cork, Ireland
e.hebrard@4c.ucc.ie

Brahim Hnich
Izmir University of Economics
Izmir, Turkey
brahim.hnich@ieu.edu.tr

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

September 11, 2006

Abstract

Constraint propagation is one of the techniques central to the success of constraint programming. To reduce search, fast algorithms associated with each constraint prune the domains of variables. With global (or non-binary) constraints, the cost of such propagation may be much greater than the quadratic cost for binary constraints. We therefore study the computational complexity of reasoning with global constraints. We first characterise a number of important questions related to constraint propagation. We show that such questions are intractable in general, and identify dependencies between the tractability and intractability of the different questions. We then demonstrate how the tools of computational complexity can be used in the design and analysis of specific global constraints. In particular, we illustrate how computational complexity can be used to determine when a lesser level of local consistency should be enforced, when constraints can be safely generalized, when decomposing constraints will reduce the amount of pruning, and when combining constraints is tractable.

1 Introduction

Constraint programming is a very successful technology for solving many kinds of combinatorial problems arising in industrial applications, such as scheduling, resource allocation, vehicle routing, and product configuration [38]. One of its key features is *constraint propagation* where values in the domains of variables are removed which will lead to a constraint violation. Constraint propagation can prune large parts of the search space, and is vital for solving combinatorially challenging problems. The notion of local consistency provides a formal way to characterise the amount of work done by constraint propagation. The most common level of local consistency, called generalised arc consistency (GAC), specifies that all values inconsistent with a constraint are pruned.

Constraint propagation on binary (or bounded arity) constraints is polynomial. However, constraint toolkits support an increasing number of global (or non-binary) constraints since such constraints are central to the success of constraint programming. See, for example, [28, 29, 8, 31, 5, 18]. Global constraints specify patterns that occur in many problems, and use constraint propagation algorithms that exploit their precise semantics. They permit users to model problems compactly and solvers to prune the search space effectively. They often allow efficient propagation. For instance, we often have sets of variables which must take different values (e.g. activities in a scheduling problem requiring the same resource must all be assigned different times). Most constraint solvers therefore provide a global '`AllDifferent`' constraint which is propagated efficiently and effectively [22, 28]. In many problems, the arity of such global constraints can grow with the problem size. For example, in the Golomb ruler problem (prob006 in CSPLib), the size of the `AllDifferent` constraint grows quadratically with the number of ticks on the ruler. Similarly, in the balanced incomplete block design (prob028 in CSPLib), the size of the intersection constraint between rows grows linearly with the number of blocks. Such global constraints may therefore exhibit complexities far beyond the quadratic cost for propagating binary constraints.

What then are the limits of reasoning with global constraints? In this paper, we show how the basic tools of computational complexity can be used to uncover many of the basic limits. We characterise the different reasoning tasks related to constraint propagation. For example, "is this value consistent with this constraint?" or "do there exist values consistent with this constraint?". We identify dependencies between the tractability and intractability of these different questions. We show that all of them are intractable in general. We therefore need to focus on specific constraints like the `AllDifferent` constraint which are tractable. We then show how these same tools of computational complexity can be used to analyse specific global constraints proposed in the past like the number of values constraint [25], as well as to help design new global constraints.

Computational complexity provides a methodology to decide when a lesser level of propagation should be enforced or when decomposing a constraint hinders propagation. It also tells us whether a new global constraint designed as a combination of elementary constraints or as a generalisation of an existing tractable constraint will itself be tractable.

The rest of the paper is organised as follows. Section 2 presents the technical background necessary to read the subsequent sections. Section 3 contains a theoretical study of generalised arc consistency, the central notion of local consistency used when speaking of constraint propagation. In Section 4, we show how the tools of computational complexity can be used to analyse different types of global constraints. An extension to meta-constraints (constraints that must be satisfied a given number of times) is presented in Section 5. Finally Section 6 discusses related work and Section 7 concludes the paper.

2 Theoretical Background

A *constraint satisfaction problem* (CSP) consists of a set of variables, each with a finite domain of values, and a set of constraints that specify allowed combinations of values for subsets of variables. We will denote variables with upper case letters and values with lower case. We will assume that the domain of a variable is given extensionally, but that a constraint C is given intensionally by a function of the form $f_C : D(X_1) \times \ldots \times D(X_n) \mapsto \{true, false\}$ where $D(X_i)$ are the domains of the variables in the scope $var(C) = (X_1, \ldots, X_n)$ of the constraint C. We say that D is a domain on $var(C)$. We cannot permit an arbitrary sort of function. For example, suppose $f_C(3, 1, 5, 2, 3, 1, \ldots)$ returns *true* iff the 1IN3-3SAT problem, $x_3 \vee x_1 \vee x_5, x_2 \vee x_3 \vee x_1, \ldots$ is satisfiable. Testing if an assignment satisfies this non-binary constraint is then NP-complete, and finding a satisfying assignment is PSPACE-complete. As a second example, suppose domains are integers of size m and $f_C(X_1, X_2, X_3, \ldots)$ is the function that halts iff $X_1 + X_2 * m + X_3 * m^2 + \ldots$ is the Gödel number of a halting Turing machine. Even testing if an assignment satisfies such a constraint is undecidable. We therefore insist that f_C is computable in polynomial time.

Constraint toolkits usually contain a library of predefined *constraint types* with a particular semantics that can be applied to sets of variables with varying arities and domains. A constraint is only an instance of a constraint type on given variables and domains. For instance, `AllDifferent` is a constraint type. `AllDifferent`$(X_1, .., X_3)$ with $D(X_1) = D(X_2) = \{1, 2\}, D(X_3) = \{1, 2, 3\}$ is an instance of constraint of the type `AllDifferent`. When there is no ambiguity, we will use the terms 'constraint' or 'constraint type' indifferently.

A solution to a CSP is an assignment of values to the variables satisfying the constraints. To find such solutions, we often use tree search algorithms that construct partial assignments and enforce a local consistency to prune the search space. Enforcing a local consistency is traditionally called *constraint propagation*. One of the most commonly used local consistencies is generalised arc consistency. A constraint C is *generalised arc consistent* (GAC) iff, when a variable in the scope of C is assigned any value in its domain, there exists an assignment to the other variables in C such that C is satisfied [24]. This satisfying assignment is called *support* for the value. On binary constraints (those involving just two variables), generalised arc consistency is called arc consistency (AC).

Since this paper makes significant use of computational complexity theory, we very briefly recall the basic tools for showing intractability. P is the class of decision problems that can be solved by a deterministic Turing machine in polynomial time, and NP is the class of decision problems that can be solved by a non-deterministic Turing machine in polynomial time. As in [19], a *trans-*

formation from a decision problem $Q_1 \in$ NP to a decision problem $Q_2 \in$ NP is a function φ that polynomially rewrites any input x of Q_1 into an input $\varphi(x)$ of Q_2 such that $Q_2(\varphi(x))$ answers "yes" if and only if $Q_1(x)$ answers "yes". If Q_1 is NP-complete, this transformation into Q_2 permits us to deduce that Q_2 is also NP-complete. A *reduction* from a problem Q_1 to a problem Q_2 (not necessarily decision problems) is a program that solves Q_1 in polynomial time under the condition that the program can call an *oracle* that solves Q_2 at most a constant number of times. If Q_1 is NP-complete, this reduction to Q_2 permits to deduce that Q_2 is NP-hard. Any NP-complete problem is thus NP-hard. By transitivity of the reductions, if Q_1 is an NP-hard problem, its reduction to Q_2 permits us to deduce that Q_2 is NP-hard. We will use *intractability* as a general term to denote any NP-hard problem, i.e., those which cannot be solved in polynomial time unless P=NP. In the following, we assume P$\neq$ NP. A problem in coNP is simply a problem in NP with the answers "yes" and "no" reversed. For instance, 3SAT, the problem of deciding if a set of ternary clauses is satisfiable, is in NP. Hence, UN3SAT, the problem of deciding if a set of ternary clauses is unsatisfiable, is in coNP. The D^P complexity class contains problems which are the conjunction of a problem in NP and one in coNP [26]. A problem Q is in D^P if there exist a NP problem Q_1 and a coNP one Q_2 such that Q answers "yes" iff Q_1 and Q_2 answer "yes". If Q_1 is NP-complete and Q_2 is coNP-complete, then Q is D^P-complete. The class D^P is also known as the second level of the Boolean hierarchy, BH_2. A typical example of a D^P-complete decision problem is the EXACT TRAVELING SALESPERSON PROBLEM where we ask if k is the length of the *shortest* tour.

3 Complexity of Generalised Arc Consistency

There are different questions that may arise when we consider enforcing generalised arc consistency. We can ask whether a value belongs to a consistent tuple or whether a constraint is generalised arc consistent. Some of the questions are more of an academic nature while others are at the heart of propagation algorithms. In this section, we formally characterise five questions related to GAC. We study the complexity of GAC reasoning on global constraints by showing intractability of two of these five questions. Finally, we show some dependencies between the intractability of the questions, from which we conclude that all five questions are intractable in general.

3.1 Questions related to GAC

We characterise five different questions related to reasoning about generalized arc consistency. These questions can be adapted to any other local consistency as long as it rules out values in domains (e.g., bounds consistency, singleton arc consistency, etc.) and not non-unary tuples of values (e.g., path consistency, relational-k-consistency, etc.)

In the following, PROBLEM($\mathcal{C}$) represents the class of questions defined by

PROBLEM on constraints of the type $\mathcal{C}$. PROBLEM($\mathcal{C}$) will be written PROBLEM when it is not confusing or when there is no restriction to a particular type of constraints. Note also that we use the notation PROBLEM[data] to refer to the instance of PROBLEM($\mathcal{C}$) with the input 'data'.

The first question we consider is at the core of all generic arc consistency algorithms. This is the question which is generally asked for all values one by one.

GACSUPPORT($\mathcal{C}$)
Instance. A constraint C of type $\mathcal{C}$, a domain D on $var(C)$, and a value v for variable X in $var(C)$
Question. Does value v for X have a support on C in D?

The second question has both practical and theoretical importance. If enforcing GAC on a particular global constraint is very expensive, we may first test whether it is necessary or not to launch the propagation algorithm (i.e., whether the constraint is already GAC). On a more academic level, this question is also commonly asked to compare different levels of local consistency.

ISITGAC($\mathcal{C}$)
Instance. A constraint C of type $\mathcal{C}$, a domain D on $var(C)$
Question. Does GACSUPPORT$[C, D, X, v]$ answer "yes" for each variable $X \in var(C)$ and each value $v \in D(X)$?

The third question can be used to decide if we do not need to backtrack at a given node in the search tree. Note that $D' \subseteq D$ stands for: $\forall X_i \in var(C), D'(X_i) \subseteq D(X_i)$.

NOGACWIPEOUT($\mathcal{C}$)
Instance. A constraint C of type $\mathcal{C}$, a domain D on $var(C)$
Question. Is there any non empty $D' \subseteq D$ on which ISITGAC$[C, D']$ answers "yes"?

An algorithm like GAC-Schema [8] removes values from the initial domain of variables till we have the (unique) *maximal* generalised arc consistent subdomain. That is, the subdomain that is GAC and any larger subdomain is not GAC. The following question characterises this "maximality" problem:

MAXGAC($\mathcal{C}$)
Instance. A constraint C of type $\mathcal{C}$, a domain D_0 on $var(C)$, and a subdomain $D \subseteq D_0$
Question. Is it the case that ISITGAC$[C, D]$ answers "yes" and there does not exist any domain D', $D \subset D' \subseteq D_0$, on which ISITGAC$[C, D']$ answers "yes"?

We finally consider the problem of returning the domain that a GAC algorithm computes. This is not a decision problem as it computes something other than "yes" or "no".

GACDOMAIN($\mathcal{C}$)
Instance. A constraint C of type $\mathcal{C}$, a domain D_0 on $var(C)$
Output. The domain D such that MAXGAC[C, D_0, D] answers "yes"

The next subsection shows the intractability of two of the above questions.

3.2 Intractability of GAC reasoning

We consider two representative decision problems at the heart of reasoning with global constraints. We will show later that their intractability implies intractability of the three others. The first is GACSUPPORT, the problem of deciding if a value for a variable has support on a constraint. In general, this is NP-complete to decide.

Observation 1 GACSUPPORT *is NP-complete.*

Proof. Clearly it is in NP as a support is a polynomial witness which can be checked (by our definition of constraint) in polynomial time. To show completeness, we transform the satisfiability of the Boolean formula φ into the problem of determining if a particular value has support. We simply construct the global constraint C involving the variables of φ plus an additional new variable X, and defined by $f_C = (X \to \varphi)$. If $X = true$ has support then φ is satisfiable. □

The second decision problem we consider is ISITGAC. Given a constraint and domains for its variables, this is the problem of deciding if these domains are GAC. This is again a NP-complete problem.

Observation 2 ISITGAC *is NP-complete.*

Proof. Clearly it is again in NP as a support for each value is a polynomial witness which can be checked in polynomial time since there are nd values involved where n is the number of variables and d the size of the largest domain. To show completeness, we transform 3COL, the problem of deciding whether a graph is 3-colorable into the problem of deciding if a particular domain is GAC for a given constraint. We introduce a variable for each vertex with domain $\{r, g, b\}$. We then define a global constraint as follows. For each pair (x_i, x_j) of vertices with an edge between in the graph, we permit pairs of values that are different (i.e., the set $\{(r, g), (r, b), (g, r), (g, b), (b, r), (b, g)\}$). For each pair (x_i, x_j) of vertices with no edge between in the graph, we permit any pair of values (i.e., the set $\{r, g, b\} \times \{r, g, b\}$). Since values are completely interchangeable, r, g and b are GAC for a variable iff the graph is 3-colorable. Hence, $\{r, g, b\}$ is a GAC domain for each variable iff the graph is 3-colorable. □

We have proven that two of the questions related to generalised arc consistency are intractable in general. In the following, we see that there are dependencies between the intractability of the five questions. This permits us to deduce that all five questions are in fact intractable in general.

3.3 Intractability relationships

The five problems defined in Section 3.1 are not independent. Knowledge about intractability of one of them can give information on intractability of others. We identify here the dependencies between intractability of the different questions.

Lemma 1 GACSupport($\mathcal{C}$) *is NP-hard iff* NoGACWipeOut($\mathcal{C}$) *is NP-hard.*

Proof. ($\Rightarrow$) GACSupport($\mathcal{C}$) can be transformed in NoGACWipeOut($\mathcal{C}$): Given $C \in \mathcal{C}$, GACSupport$[C, D, X, v]$ is solved by calling NoGACWipeOut$[C, D|_{D(X)=\{v\}}]$.

($\Leftarrow$) NoGACWipeOut$[C, D]$ can be reduced to GACSupport by calling GACSupport$[C, D, X, v]$ for each value v in $D(X)$ for one of the X in $var(C)$. GAC leads to a wipe out iff none of these values has a support. □

Lemma 2 GACSupport($\mathcal{C}$) *is NP-hard iff* GACDomain($\mathcal{C}$) *is NP-hard.*

Proof. ($\Rightarrow$) GACSupport($\mathcal{C}$) can be reduced to GACDomain($\mathcal{C}$) since GACSupport$[C, D, X, v]$ answers "yes" iff GACDomain$[C, D|_{D(X)=\{v\}}]$ doesn't return an empty domain.

($\Leftarrow$) GACDomain$[C, D]$ can be reduced to GACSupport by performing a polynomial number of calls to GACSupport$[C, D, X, v]$, one for each $v \in D(X)$, $X \in var(C)$. When the answer is "no" the value v is removed from $D(X)$, otherwise it is kept. The domain obtained at the end of this process represents the output of GACDomain. □

Corollary 1 NoGACWipeOut($\mathcal{C}$) *is NP-hard iff* GACDomain($\mathcal{C}$) *is NP-hard.*

Lemma 3 *If* maxGAC($\mathcal{C}$) *is NP-hard then* GACSupport($\mathcal{C}$) *is NP-hard.*

Proof. maxGAC$[C, D_0, D]$ can be reduced to GACSupport. We perform a polynomial number of calls to GACSupport$[C, D_0, X, v]$, one for each $v \in D_0(X)$, $X \in var(C)$. When the answer is "yes" the value v is added to a (initially empty) set $D'(X)$. maxGAC answers "yes" if and only if the domain D' obtained at the end of the process is equal to D. □

Lemma 4 *If* IsItGAC($\mathcal{C}$) *is NP-hard then* maxGAC($\mathcal{C}$) *is NP-hard.*

Proof. IsItGAC$[C, D]$ can easily be transformed into maxGAC$[C, D, D]$. □

Corollary 2 *If* IsItGAC($\mathcal{C}$) *is NP-hard then* GACSupport($\mathcal{C}$) *is NP-hard.*

It is worth noting that whilst intractability of IsItGAC implies that of maxGAC, this last question may be outside NP. In fact, maxGAC is D^P-complete.

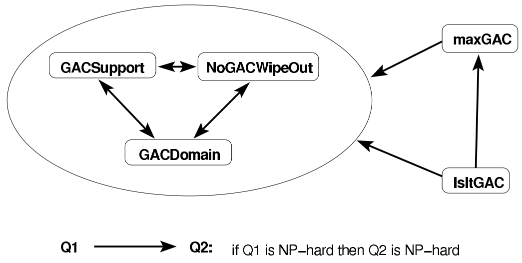

Figure 1: Summary of dependencies between intractable problems

Theorem 1 MaxGAC *is D^P-complete.*

Proof. A problem Q is D^P-complete if there exist Q_1 and Q_2 such that Q_1 is NP-complete, Q_2 is coNP-complete, and Q answers "yes" iff Q_1 and Q_2 answer "yes". We use 3Col and Un3Col as Q_1 and Q_2. We suppose without loss of generality that Q_1 and Q_2 both involve the same set X of vertices. E_i is the set of edges in Q_i.

We introduce a variable for each vertex with domain $\{r_1, g_1, b_1, r_2, g_2, b_2\}$. We then define a global constraint as follows. For each pair (x_i, x_j) of vertices with an edge between in both Q_1 and Q_2, we permit pairs of values that are different but have the same subscript (i.e., the set $\{(r_1, g_1)$, (r_1, b_1), (g_1, r_1), (g_1, b_1), (b_1, r_1), (b_1, g_1), (r_2, g_2), (r_2, b_2), (g_2, r_2), (g_2, b_2), (b_2, r_2), $(b_2, g_2)\}$). For each pair (x_i, x_j) of vertices with an edge between in Q_1 and not in Q_2, we permit pairs of values that are different for the subscript 1, and any combination for subscript 2 (i.e., the set $\{(r_1, g_1), (r_1, b_1), (g_1, r_1), (g_1, b_1), (b_1, r_1), (b_1, g_1)\} \cup \{r_2, g_2, b_2\} \times \{r_2, g_2, b_2\}$). Similarly, for each pair (x_i, x_j) of vertices with an edge between in Q_2 and not in Q_1, we permit pairs of values that are different for the subscript 2, and any combination for subscript 1. Finally, for each pair (x_i, x_j) of vertices with no edge between in Q_1 or in Q_2, we permit any pairs of values with the same subscript (i.e., the set $\{r_1, g_1, b_1\} \times \{r_1, g_1, b_1\} \cup \{r_2, g_2, b_2\} \times \{r_2, g_2, b_2\}$). By construction, r_i, g_i and b_i are GAC iff (X, E_i) is 3-colorable. Hence, $\{r_1, g_1, b_1\}$ is the maximal GAC subdomain for each variable iff (X, E_1) is 3-colorable, and (X, E_2) is not 3-colorable. $\square$

A summary of the dependencies proved in Lemmas 1–4 and Corollary 1 is given in Fig. 1. Note that since each arrow from question A to question B in Fig. 1 means that A can be rewritten as a polynomial number of calls to B, we immediately have that tractability of B implies tractability of A. (See Fig. 2 for tractability dependencies of the five questions.)

Reasoning with global constraints is thus not tractable in general. Global constraints which are used in practice are therefore usually part of that special subset for which constraint propagation is polynomial. For example, GAC on an n-ary `AllDifferent` constraint can be enforced in $O(n^{\frac{3}{2}}d)$ time [28]. In the rest of this paper, we show how we can further use the tools of computational complexity in the design and analysis of *specific* global constraints.

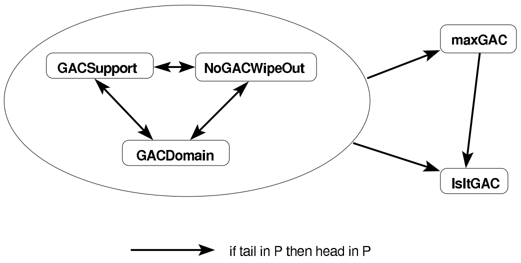

Figure 2: Summary of the dependencies between tractable problems

4 Using Intractability Results

The tools of computational complexity can also be used to analyse existing global constraints for which no polynomial algorithm is known, or can help us in designing new global constraints for specific purposes. To prove that a constraint type $\mathcal{C}$ is intractable, we generally transform/reduce some known NP-complete/NP-hard problem to the existence of a satisfying assignment for $\mathcal{C}$, i.e., the NoGACWipeOut($\mathcal{C}$) problem. Thanks to the dependency results shown above, we can then deduce intractability of GACSupport and GACDomain. For the more academic questions, IsItGAC and maxGAC, the complexity cannot be deduced from our dependencies since they are 'exact' problems (a "no" answer brings little information). Finally, we sometimes do not need the full expressive power of a constraint type to prove its intractability. For example, we may use only a fixed value for one of the variables involved in the constraint. In this case, the constraint is also intractable if we use its full expressive power.

We can derive several kinds of information about global constraints by using computational complexity results. For example, on existing global constraints for which no polynomial algorithm is known for a given level of local consistency, proving intractability tells us that no such algorithm exists, and that we should look to enforce a lesser level of consistency. On constraints that decompose into simpler constraints which have polynomial propagation algorithms, intractability results not only tell us that this decomposition hinders propagation, but that there cannot exist any decomposition on which we achieve GAC in polynomial time. We also sometimes want to use an already existing global constraint in a form more general than its original definition. A proof of intractability tells us that generalisation makes the constraint impossible to propagate in polynomial time.

The remainder of this section gives examples of existing and new global constraints that we analyse with these tools of computational complexity.

4.1 Local consistency

Computational complexity results can indicate what level of local consistency to enforce on a constraint. If achieving a given local consistency on a constraint is NP-hard, then enforcing a lower level of consistency may be advisable. For example, the number of values constraint, $\texttt{NValue}(X_1, \ldots, X_n, N)$ [25, 3] ensures that N distinct values are used by the n finite domain variables X_i. Note that N can itself be an integer variable. The `AllDifferent` constraint is a special case of the `NValue` constraint in which $N = n$. The `NValue` constraint is useful for reasoning about resources. For example, if the values are workers assigned to a particular shift, we may have a `NValue` constraint on the number of workers that a set of shifts can involve. Whilst there exists an $O(n^{2.5})$ algorithm for enforcing GAC on the `AllDifferent` constraint [28], enforcing GAC on the `NValue` constraint is intractable in general.

Theorem 2 *Enforcing GAC on a* $\texttt{NValue}(X_1, \ldots, X_n, N)$ *constraint is NP-hard, and remains so even if* N *is ground and different to* n.

Proof. We use a transformation from 3SAT to NoGACWipeOut(`NValue`). Given a 3SAT problem in n variables (labelled from 1 to n) and m clauses, we construct the constraint $\texttt{NValue}([X_1, \ldots, X_{n+m}], N)$ in which $D(X_i) = \{i, -i\}$ for all $i \in [1, n]$, and each X_i for $i > n$ represents one of the m clauses. If the jth clause is $x \vee \neg y \vee z$ then $D(X_{n+j}) = \{x, -y, z\}$. The constructed constraint where $D(N) = \{n\}$ has a solution iff the original 3SAT problem has a satisfying assignment. Hence deciding if enforcing GAC on `NValue` does not lead to a domain wipe out is NP-complete, and enforcing GAC is itself NP-hard. □

If we want to maintain a reasonable cost for propagation, we therefore probably have to enforce a lower level of consistency. For instance, there exists a polynomial algorithm for enforcing bound consistency on the `NValue` constraint [7].

As a second example, let us take the `Common` constraint, $\texttt{Common}(N, M, [X_1, \ldots, X_n], [Y_1, \ldots, Y_m])$ introduced in [1], that ensures that $N = |\{i \mid \exists j,\ X_i = Y_j\}|$ and $M = |\{j \mid \exists i,\ X_i = Y_j\}|$. That is, N is the number of variables in the X_i that take values in the Y_j, and M is the number of variables in the Y_j that take values in the X_i. The `AllDifferent` constraint is again a special case of the `Common` constraint in which the Y_j enumerate all the values j in the X_i, $Y_j = \{j\}$ and $M = n$.

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

Proof. In Theorem 8, it is shown that enforcing GAC on $\texttt{Among}(N, [X_1, \ldots, X_n], [D_1, \ldots, D_m])$ is NP-hard, where the constraint holds iff $N = |\{i \mid \exists j, X_i = D_j\}|$. Deciding if enforcing GAC on such an `Among` constraint does not lead to a domain wipe out is equivalent to deciding if enforcing GAC on $\texttt{Common}(N, M, [X_1, \ldots, X_n], [D_1, \ldots, D_m])$ with $D(M) = \{0, \ldots, n\}$ does not lead to a domain wipe out. As a result, enforcing GAC on `Common` is itself NP-hard. □

4.2 Decomposing constraints

Computational complexity results can tell us more than just what level of local consistency to enforce. It can also indicate properties that any possible decomposition of a constraint must possess. We say that a decomposition of a global constraint is *GAC-poly-time* if we can enforce GAC on the decomposition in time polynomial in the size of the original constraint and domains. The following lemma tells us when such decomposition hinders constraint propagation.

Lemma 5 *If enforcing GAC on a constraint C is NP-hard, then there does not exist any GAC-poly-time decomposition of C that achieves GAC on C.*

Proof. By definition, enforcing GAC on a GAC-poly-time decomposition is polynomial. Hence, if GAC on the decomposition was equivalent to GAC on the original constraint, then P would equal NP. □

Consider a constraint that ensures two sequences of variables are disjoint (i.e. have no value in common). For example, two sequences of tasks sharing the same resource might be required to be disjoint in time. The `Disjoint`($[X_1, \ldots, X_n]$, $[Y_1, \ldots, Y_m]$) constraint introduced in [1] ensures $X_i \neq Y_j$ for any i and j. This constraint has a very simple and natural decomposition into the set of all binary constraints $X_i \neq Y_j, i \in [1, n], j \in [1..m]$. Unfortunately, enforcing AC on this decomposition into binary constraints does not achieve GAC on the corresponding `Disjoint` constraint. Consider $X_1, Y_1 \in \{1, 2\}$, $X_2, Y_2 \in \{1, 3\}$, and $Y_3 \in \{2, 3\}$. The decomposition into binary constraints is already AC. However, enforcing GAC on `Disjoint`($[X_1, X_2], [Y_1, Y_2, Y_3]$) prunes 3 from X_2 and 1 from both Y_1 and Y_2.

Moreover, we prove here that we cannot expect any decomposition to achieve GAC on such a constraint.

Theorem 4 *GAC on any GAC-poly-time decomposition of the* `Disjoint` *constraint is strictly weaker than GAC on the undecomposed constraint.*

Proof. We show that enforcing GAC on a `Disjoint` constraint is NP-hard, and then appeal to Lemma 5. We reduce 3SAT to NOGACWIPEOUT(`Disjoint`). Consider a formula φ with n variables and m clauses. We let $X_i \in \{i, -i\}$ and $Y_j \in \{x, -y, z\}$ where the jth clause in φ is $x \vee \neg y \vee z$. If φ has a model then the `Disjoint` constraint has a satisfying assignment in which the X_i take the literals false in this model and the Y_j take the literal satisfying the jth clause. Hence, deciding if enforcing GAC does not lead to a domain wipe out on `Disjoint` is NP-complete, and enforcing GAC is itself NP-hard. □

As another example, Sadler and Gervet introduce the `AtMost1` constraint [32]. This ensures that n set variables of a fixed cardinality c intersect in at most one value. To fit this within the theoretical framework presented in this paper, we consider the characteristic function representation for each set variable (i.e. a vector of 0/1 decision variables). Enforcing GAC on such a representation is equivalent to enforcing bounds consistency on the upper and lower bounds of the set variables [39]. The `AtMost1` constraint can be decomposed into pairwise intersection and cardinality constraints. That is, it can be decomposed into $|X_i \cap X_j| \leq 1$ for $i < j$ and $|X_i| = c$ for all i. On the characteristic function representation, this is $\sum_k X_{ik} \cdot X_{jk} \leq 1$ and $\sum_k X_{ik} = c$, which are both GAC-poly-time. Such decomposition hinders constraint propagation.

Theorem 5 *GAC on any GAC-poly-time decomposition of the* `AtMost1` *constraint is strictly weaker than GAC on the undecomposed constraint.*

Proof. We show that enforcing GAC on an `AtMost1` constraint is NP-hard, and appeal to Lemma 5. To show that enforcing GAC on the `AtMost1` constraint is NP-hard, we consider the case when the cardinality $c = 2$. For $c > 2$, we can use a similar construction as in the $c = 2$ reduction but add $c - 2$ distinct values to each set. The proof uses a reduction from 3SAT. For each clause σ, we introduce a set variable, X_σ. Suppose $\sigma = x_i \vee \neg x_j \vee x_k$, then X_σ has the domain $\{m_\sigma\} \subseteq X_\sigma \subseteq \{m_\sigma, i_\sigma, \neg j_\sigma, k_\sigma\}$. If the intersection and cardinality constraint is satisfied, X_σ takes the value $\{m_\sigma, i_\sigma\}$, $\{m_\sigma, \neg j_\sigma\}$, or $\{m_\sigma, k_\sigma\}$. The first case corresponds to x_i being *true* (which satisfies σ), the second to $\neg x_j$ being *true*, and the third to x_k being *true*.

We use an additional (at most quadratically many) set variables to ensure that contradictory assignments are not made to satisfy other clauses. Suppose we satisfy σ by assigning x_i to *true*. That is, $X_\sigma = \{m_\sigma, i_\sigma\}$. Consider any other clause, τ which contains $\neg x_i$. We construct two set variables, $Y_{\sigma\tau i}$ and $Z_{\sigma\tau i}$ with domains $\{m_\sigma\} \subseteq Y_{\sigma\tau i} \subseteq \{m_\sigma, i_\sigma, \neg i_\tau\}$ and $\{\neg i_\tau\} \subseteq Z_{\sigma\tau i} \subseteq \{m_\sigma, m_\tau, \neg i_\tau\}$. Since $X_\sigma = \{m_\sigma, i_\sigma\}$, then $Y_{\sigma\tau i} = \{m_\sigma, \neg i_\tau\}$ and $Z_{\sigma\tau i} = \{m_\tau, \neg i_\tau\}$. Hence, $X_\tau \neq \{m_\tau, \neg i_\tau\}$. That is, τ cannot be satisfied by $\neg x_i$ being assigned *true*. Some other literal in τ has to satisfy the clause.

The constructed set variables thus have a solution which satisfies the intersection and cardinality constraints iff the original 3SAT problem is satisfiable. Hence deciding if enforcing GAC on `AtMost1` does not lead to a domain wipe out is NP-complete, and enforcing GAC is itself NP-hard. □

A similar result can be given for the `Distinct` constraint introduced in [32]. This constraint ensures that n set variables of a fixed cardinality intersect in at least one value. Again, a GAC-poly-time decomposition of such a constraint hinders constraint propagation.

4.3 Combining constraints

Global constraints specify patterns that reoccur in many problems. However, there may only be a limited number of common constraints which repeatedly occur in problems. One strategy for developing new global constraints is to identify conjunctions of constraints that often occur together, and developing constraint propagation algorithm for their combination. For example, [31] propose a propagation algorithm for a constraint which combines together sum and difference constraints. As a second example, [10] combine together a chain of lexicographic ordering constraints. As a third example, [21] combine together a lexicographic ordering and two sum constraints.

We can use results from computational complexity to determine when we should not combine together constraints. For example, scalar product constraints occur in many problems like the balanced incomplete block design, template design and social golfers problems [40]. Often such problems have scalar product constraints between all pairs of rows in a 2-dimensional array of Boolean decision variables. We can enforce GAC on a scalar product constraint between two rows in linear time. Should we consider combining together all the row scalar product constraints into one large global constraint? Such a `ScalarProduct` constraint would ensure that $\forall i < j \ \sum_k X_{ik} \cdot X_{jk} = p$. The following result shows that enforcing GAC on such a composition of constraints is intractable.

Theorem 6 *Enforcing GAC on a* `ScalarProduct` *constraint is NP-hard, even when restricted to 0/1 variables.*

Proof. We consider the case when the scalar product $p = 1$. For $p > 1$, we use a reduction that adds $p-1$ additional columns to the array, each column containing variables that must take the value 1.

We reduce 1IN3-3SAT on positive formulae (which is NP-complete [19]) to deciding if enforcing GAC does not lead to a domain wipe out on a `ScalarProduct` constraint over 0/1 variables. Given a 1IN3-3SAT problem in n variables and m clauses, we construct a `ScalarProduct` constraint with $4m+1$ rows and $3m+n$ columns. The first row of the array, where all variables have 0/1 domain, represents the model which satisfies the 1IN3-3SAT problem. There is a column for each occurrence of a literal in a clause. That is, the $(3(j-1)+k)$th column represents the kth literal in the jth clause. This is assigned 1 in the first row iff the corresponding literal is *true*. There is also a column for the negation of each literal. That is, the $(3m+i)$th column represents the negation of the ith literal. This is assigned 1 in the first row iff the corresponding literal is *false*.

The remaining rows are divided into two types. First, there is a row for each clause. In the $(1+j)$th row, representing the jth clause, the columns $3(j-1)+1, 3(j-1)+2, 3(j-1)+3$ corresponding to literals in the clause have value 1. The other columns have the value 0. The scalar product constraint between a row representing a clause and the row representing the model ensures that only one of the literals in the clause is *true*. Second, there are rows for each occurrence of a positive literal to ensure that the row representing the model does not assign both a literal and its negation to *true*. That is, if the ith variable of the formula appears as the kth literal in the jth clause, then, in the $(1+m+3(j-1)+k)$th row, the columns $3(j-1)+k$ and $3m+i$ have value 1. The other columns have the value 0.

The 1IN3-3SAT problem has a model iff the constructed array has a solution. Hence deciding if enforcing GAC does not lead to a domain wipe out on `ScalarProduct` is NP-complete, and enforcing GAC is NP-hard. □

Special cases of the `ScalarProduct` constraint are tractable. For instance, if the scalar product is zero and variables are 0/1 then the constraint is equivalent to the pairwise Disjoint constraint on set variables, which is tractable [39].

4.4 Generalising constraints

Another way in which tools of computational complexity can help is when we generalise existing global constraints. We might have a global constraint with a polynomial propagation algorithm, but want to use it in a more general manner. For example, we might want to replace a given constant parameter with a variable or to repeat the same variable several times in the scope of the constraint.

4.4.1 Constant parameter becoming a variable

The global cardinality constraint, $\texttt{Gcc}([X_1, \dots, X_n], [O_1, \dots, O_m])$, ensures that $O_j = |\{i \mid X_i = j\}$ for all j. That is, the value j occurs O_j times in the variables X_i. The special case of this constraint where O_j are fixed intervals was presented in [29] together with a polynomial propagation algorithm enforcing GAC on the X_i. The `AllDifferent` constraint is a special case of the `Gcc` constraint in which $O_j = [0, 1]$. However, to enforce GAC on the more general form of the `Gcc` constraint where the O_j are integer variables is NP-hard.

Theorem 7 ([27]) *Enforcing GAC on a* $\texttt{Gcc}([X_1, \dots, X_n], [O_1, \dots, O_m])$ *where the* O_j *are integer variables is NP-hard.*

A second example is the `Among` constraint. The $\texttt{Among}(N, [X_1, \dots, X_n], [d_1, \dots, d_m])$ constraint, introduced in CHIP [5] ensures that $N = |\{i \ / \ \exists j, X_i = d_j\}|$. That is, N variables in X_i take values in $[d_1, \dots, d_m]$. The `Among` constraint is a generalisation of the `AtMost` and `AtLeast` constraints. Enforcing GAC is polynomial on the `Among` constraint. A generalisation of this constraint is to let the d_j be integer variables D_j instead of constants. In this case, enforcing GAC becomes intractable.

Theorem 8 *Enforcing GAC on* $\texttt{Among}(N, [X_1, \dots, X_n], [D_1, \dots, D_m])$ *is NP-hard.*

Proof. We again use a transformation from 3SAT. Given a 3SAT problem in n variables (labelled from 1 to n) and m clauses, we construct the `Among` constraint, $\texttt{Among}(N, [X_1, \dots, X_m], [D_1, \dots, D_n])$ in which $D(N) = \{m\}$, $D(D_i) = \{i, -i\}$, and each X_j represents one of the m clauses. If the jth clause is $x \vee \neg y \vee z$ then $D(X_j) = \{x, -y, z\}$. The constructed `among` constraint has a solution iff the original 3SAT problem has a model. Hence deciding if enforcing GAC does not lead to a domain wipe out is NP-complete, and enforcing GAC is itself NP-hard. □

4.4.2 Repeating variables

In the constraint $\texttt{Gcc}([X_1, \ldots, X_n], [O_1, \ldots, O_m])$, the number of occurrences O_j for a value j is a fixed interval $[l_j..u_j]$. In addition, we assume that no variables in the sequence $[X_1, \ldots, X_n]$ are repeated. However, there are problems in which we would like to have a gcc constraint with the same variable occurring several times in $[X_1, \ldots, X_n]$, or equivalently, some variables that must take the same value. For example, in shift rostering, we might have constraints on the number of shifts worked by each individual, as well as the requirement that the same person works consecutive weekends. This can be modelled with a Gcc with repeated variables. Unfortunately, achieving arc consistency (GAC) on Gcc with repeated variables is intractable.

Theorem 9 *Enforcing GAC on a* $\texttt{Gcc}([X_1, \ldots, X_n], [O_1, \ldots, O_m])$ *where variables in* $[X_1, \ldots, X_n]$ *can be repeated is NP-hard even if the* O_j *are fixed intervals.*

Proof. We transform 3SAT into NoGACWipeOut(Gcc). Let $\varphi = \{c_1, \ldots, c_m\}$ be a 3CNF on the Boolean variables $x_1, \ldots, x_n$. We build the constraint $\texttt{Gcc}(Y, [O_{-n}, \ldots, O_{-1}, O_1, \ldots, O_n])$ where:

1. $Y = [Y_{c_1}, \ldots, Y_{c_m}, Y_{l_1}^{(1)}, \ldots, Y_{l_1}^{(m)}, Y_{l_2}^{(1)}, \ldots, Y_{l_n}^{(m)})]$, where $Y_{l_i}^{(1)}, \ldots, Y_{l_i}^{(m)}$ are m copies of the same variable Y_{l_i} with $D(Y_{l_i}) = \{i, -i\}$ and $D(Y_{c_j}) = \{j_1, -j_2, j_3\}$ if $c_j = x_{j_1} \vee \neg x_{j_2} \vee x_{j_3}$,
2. $O_i = [0, m], \forall i \in [-n, -1] \cup [1, n]$,

Consider a model of φ. If x_{i_k} is one of the variables in clause c_i that make c_i true in the model, assign Y_{c_i} with i_k if x_{i_k} is true, and $-i_k$ otherwise. For every i, assign Y_{l_i} with i if x_i is false and $-i$ otherwise. This assignment is a solution for Gcc.

Consider now a solution for Gcc. Then x_i set to true iff $Y_{l_i} = -i$ is a model of φ. The m occurrences of each Y_{l_i} and the capacities O_j in the Gcc ensure that none of the Y_{c_k} can take $-i$ if $Y_{l_i} = -i$ or i if $Y_{l_i} = i$.

The constructed Gcc constraint with repeated variables has a solution iff the original 3SAT problem has a model. Hence deciding if enforcing GAC does not lead to a domain wipe out is NP-complete, and enforcing GAC is itself NP-hard. □

We see that computational complexity can tell us when we will need to enforce a lesser level of consistency on the generalisation of an existing global constraint.

5 Meta-Constraints

Computational complexity can also be used to study "meta-constraints" that combine together other constraints. We will show that even when the constraints being combined are tractable to propagate, the meta-constraint itself might not be tractable to propagate. For example, the `Card` constraint [36] is provided by many constraint toolkits. It ensures that N constraints from a given set are satisfied, where N is an integer decision variable. The most general form of the constraint is: $\texttt{Card}(N, [C_1, \ldots, C_m])$ where C_i are themselves constraints (not necessarily all of the same arity), and $N = |\{i \mid C_i \text{ is satisfied}\}|$. The cardinality constraint can be used to implement conjunction, ($C_1 \wedge C_2$ is equivalent to $\texttt{Card}(2, [C_1, C_2])$), disjunction, ($C_1 \vee C_2$ is equivalent to $\texttt{Card}(N, [C_1, C_2])$ where $N \geq 1$), negation, ($\neg C_1$ is equivalent to $\texttt{Card}(0, [C_1])$). It has had numerous applications in a wide range of domains including car-sequencing, disjunctive scheduling, Hamiltonian path and digital signal processor scheduling [37].

It is obvious that $\texttt{Card}(N, [C_1, \ldots, C_m])$ is tractable if the constraints C_i have bounded arity and do not share any variable. However, only a limited form of consistency is enforced on a `Card` constraint (see [23]), and it is easy to show why this is necessary in general.

Theorem 10 *Enforcing GAC on the* $\texttt{Card}(N, [C_1, \ldots, C_m])$ *constraint is NP-hard, and remains so even if all the constraints* C_i *are identical and binary and no variable is repeated more than three times.*

Proof. We use a reduction from the special case of 3SAT in which at most three clauses contain a variable or its negation. (This is still NP-complete.) Each Boolean variable x is represented by a CSP variable X with domain $\{0, 1\}$. Each clause σ is represented by three CSP variables, U_σ, V_σ and W_σ, and five binary constraints posted on these variables. The domain of U_σ is a strict subset of $\{8, \ldots, 15\}$, of V_σ is a strict subset of $\{16, \ldots, 23\}$ and of W_σ is a strict subset of $\{24, \ldots, 31\}$. The domain values serve two purposes. First, the bottom three bits indicate the truth values taken by the variables that satisfy the clause. We therefore have to delete one value from each domain. This is the assignment of truth values which does not satisfy the clause. For example, if σ is $x \vee \neg y \vee z$ then the only assignment to X, Y and Z, which does not satisfy the clause is 0, 1, 0. We therefore delete the value 26 from W_σ as $26 \bmod 8$ is 2 (or 010 in binary). Similarly, we delete the value 18 from V_σ as $18 \bmod 8$ is 2, and 10 from U_σ. Second, the top two bits of the values of U_σ, V_σ and W_σ point to one of the three positions in the clause. We add three binary constraints to the cardinality constraint: $C(U_\sigma, X)$, $C(V_\sigma, Y)$ and $C(W_\sigma, Z)$.

We also need to ensure that U_σ, V_σ and W_σ take consistent values. We therefore add two binary constraints: $C(U_\sigma, V_\sigma)$, and $C(V_\sigma, W_\sigma)$. Finally, we define $C(X, Y)$ as follows. If $Y \in \{0, 1\}$, there are three cases. If $8 \leq X \leq 15$ then C is satisfied iff $(X \bmod 8) \operatorname{div} 4 = Y$ (i.e., the third bit of X agrees with Y). If $16 \leq X \leq 23$ then C is satisfied iff $(X \bmod 4) \operatorname{div} 2 = Y$ (i.e., the second bit of X agrees with Y). If $24 \leq X \leq 31$ then C is satisfied iff $X \bmod 2 = Y$ (i.e., the first bit of X agrees with Y). Otherwise $Y \geq 8$ and C is satisfied iff $X \bmod 8 = Y \bmod 8$.

The constructed cardinality constraint has a solution iff there is an assignment to the Boolean variables that satisfies all of the clauses. Hence enforcing GAC is NP-hard. □

A more restricted, but nevertheless very useful form of the cardinality constraint is the cardinality path constraint [4]. The most general form of the constraint is: $\texttt{Cardpath}(N, [X_1, \ldots, X_m], C)$ where C is a constraint of arity k, and $N = |\{i \in 1..m-k+1 \mid C(X_i, \ldots, X_{i+k-1}) \text{ is satisfied}\}|$. This "slides" a constraint of fixed arity down a sequence of variables and ensures that it holds N times, where N is itself an integer decision variable. This constraint can be used to implement the change constraint, (which counts the number of changes of value in a sequence), smooth constraint (which limits the size of changes of value along a sequence), number of rests constraint (which counts the number of two day or more rests in a sequence), and sliding sum constraints. In [4], a greedy algorithm is given for propagating the cardinality path constraint. However, even for binary constraints, the algorithm fails to prune all possible values. In [6], an algorithm is proposed that achieves GAC when no variable is repeated in the sequence $[X_1, \ldots, X_m]$ and C has arity k. This takes a time which is polynomial in m but exponential in k. If k is bounded (e.g. $k = 2$), this is polynomial. The algorithm uses dynamic programming technique that slides along the values of the variables the number of times C can be satisfied in a tuple involving the given value. After two passes of this sliding process, the values from N that never appear in the counters can be pruned, as well as the values that are not labelled by any value in the domain of N. As soon as we allow repetitions of variables in the sequence, it is not hard to show that enforcing GAC on `Cardpath` is intractable. As with `Gcc`, this is another example of constraint that changes from polynomial to intractable when we allow repeated variables.

Theorem 11 *Enforcing GAC on* $\texttt{Cardpath}(N, [X_1, \ldots, X_m], C)$ *where variables in the sequence* $[X_1, \ldots, X_m]$ *can be repeated is NP-hard even if* C *is binary.*

Proof. We use a reduction from 3COL. We assume without loss of generality that the graph is connected. Each node in the graph is represented by a CSP variable. The domain of each variable is the set of three colours. We then construct a sequence of variables $X_1, \ldots, X_m$ such that if there is an edge (i, j) in the graph then there is at least one position in the sequence with X_i next to X_j. To do this, we pick any node at which to start. We then pick any edge in the graph not yet in the sequence and find a path from our starting node that passes through this edge. We add this path to our sequence. We repeat until all edges are in the sequence. Finally, we set $N = m - 1$ and C to be the binary not-equals constraint. The constructed cardinality path constraint has a solution iff there is a proper colouring of the graph. Hence deciding if enforcing GAC does not lead to a domain wipe out is NP-complete, and enforcing GAC is itself NP-hard. □

It is less easy to see that enforcing GAC on $\texttt{Cardpath}(N, [X_1, \ldots, X_m], C)$ is intractable when the sequence of variables $[X_1, \ldots, X_m]$ does not contain any repetition and GAC can be enforced on C in polynomial time.

Theorem 12 *Enforcing GAC on* $\texttt{Cardpath}(N, [X_1, \ldots, X_m], C)$ *is NP-hard even* *when enforcing GAC on C is polynomial and no variable is repeated in the sequence.*

Proof. We transform MAX2SAT into NOGACWIPEOUT(`Cardpath`). MAX2-SAT is the problem of deciding whether there exists an assignment of n Boolean variables violating at most k clauses in a 2SAT formula with m clauses. The idea is to build a sequence of variables, alternating n Boolean variables with two variables representing one of the binary clauses, and then again n Boolean variables and so on until all clauses are represented. The sliding constraint C guarantees that in each alternation, the assignment of the n Boolean variables on the left of the two clause-variables is equal to the assignment on the right (i.e. the same assignment is used down the sequence), and that the binary clause sandwiched in the middle is satisfied by this assignment. To prevent violation of a clause being confused with a change in the assignment, we need $k+1$ dummy variables in each alternation. A change in the assignment then violates $k+1$ times the constraint C. (k is the bound of the MAX2SAT problem.) So, the whole sequence is composed of m alternations, each with $k+1$ dummy variables plus n Boolean variables plus two clause-variables, plus some additional dummy variables at the very end of the sequence to guarantee that the last clause is checked. The domain of the dummy variables is $\{n+1\}$, that of Boolean variables is $\{0,1\}$. If $c_j = x_{i_1} \vee \neg x_{i_2}$, the first clause-variable in the jth alternation has domain $\{i_1\}$ and the second has domain $\{-i_2\}$. The constraint C, of arity $2(k+1)+2n+4$ (two alternations), is built to be satisfied in the three following cases: if neither its first variable nor its $(k+1+n+2)$th is a dummy ($k+1+n+2$ is the length of an alternation); if its first variable is a dummy and the two assignments of n Boolean variables are the same; finally if the first variable is not a dummy, the $(k+1+n+2)$th variable is, and the clause represented by the two variables in positions $n+1$ and $n+2$ is satisfied by the assignment. Enforcing GAC on C is clearly polynomial.

There remains to set the domain of N to the interval from the total number of occurrences of C in the sequence (all C satisfied) to this number less k. This ensures that C is violated at most k times. As a change in the assignment to the Boolean variables costs at least $k+1$ violations, we are guaranteed that the same assignment 'slides down' the sequence. Thus `Cardpath` has a satisfying tuple if and only if there exists an assignment of the Boolean variables of the MAX2SAT formula that violates at most k binary clauses. Therefore, deciding if enforcing GAC does not lead to a domain wipe out is NP-complete, and enforcing GAC is itself NP-hard. □

We have seen that `Cardpath` is tractable when C has a fixed arity, and we do not allow repetitions of variables in the sequence. However, as soon as we relax either one of these restrictions, propagation becomes NP-hard. We may therefore need to enforce a lesser level of local consistency such as in [4].

6 Related Work

Analysis of tractability and intractability is not new in constraint programming. Identifying properties under which a constraint satisfaction problem is tractable has been studied for a long time. For example, Freuder [17], Dechter and Pearl [13, 14] or Gottlob et al [20] gave increasingly general conditions on the structure of the underlying (hyper)graph to obtain a backtrack-free resolution of a problem. van Beek and Dechter [35] and Deville et al [15] presented conditions on the semantics of the individual constraints that make the problem tractable. Finally, Cohen et al [12] showed that when the constraints composing a problem are defined as disjunctions of other constraints of specified types, then the whole problem is tractable. However, these lines of research are concerned with a constraint satisfaction problem as a whole, and do not say much about individual particular constraints.

For constraints of bounded arity, asymptotic analysis has been extensively used to study the complexity of constraint propagation both in general and for constraints with a particular semantics. For example, the GAC-Schema algorithm of [8] has an $O(d^n)$ time complexity on constraints of arity n and domains of size d, whilst the GAC algorithm of [28] for the n-ary `AllDifferent` constraint has $O(n^{\frac{3}{2}}d)$ time complexity. These are upper bounds on the cost of GAC in general or on specific constraints. By comparison, we have characterised here conditions under which no polynomial algorithm for GAC can be designed for a given constraint type.

For global constraints like the `Cummulative` and `Cycle` constraints, there are very immediate reductions from the bin packing and Hamiltonian circuit which demonstrate that reasoning with these constraints is intractable in general. It is therefore perhaps not surprising that there has been little comment in the past about their intractability. However, as we show here, there are many other global constraints proposed in the past like `NValue` and `AtMost1` where a reduction is less immediate, but the constraint is intractable nevertheless.

In many constraint problems, the goal is not only to satisfy all the constraints, but also to minimise (or maximise) an objective function. Constraint propagation can be enhanced in these problems by cost-based filtering where we also remove values that are proven sub-optimal. *Optimisation constraints*, that combine a regular constraint of the problem with a constraint on the maximal value the objective function can take have been advocated in [11]. GAC on such a combined constraint will not only prune the values having no support on the regular constraint, but also the values that do not extend to any satisfying assignment of the constraint improving the given bound. However, as in the case of combining constraints (see Section 4.3), such compositions have to be handled with care. The optimisation version of a constraint for which enforcing GAC is intractable obviously remains intractable (e.g., [33]). However, the optimisation version of a constraint for which GAC is polynomial either remains tractable (e.g., [16, 30]) or may become intractable. An example of the latter situation is the shortest path constraint, which is the optimisation version of the path constraint [34].

Beldiceanu has proposed a general framework for describing many global constraints in terms of graph properties on structured networks of simple elementary constraints [2]. It is an interesting open question if we can identify properties or elementary constraints within this framework which guarantee that a global constraint is computationally (in)tractable. Finally, computational complexity can help us classify the "globality" of constraints [9]. Indeed, NP-hardness of enforcing GAC is a sufficient condition for a constraint to be *operationally GAC-global* wrt GAC-poly-time decompositions.

7 Conclusions

We have studied the computational complexity of reasoning with global constraints. We have considered a number of important questions related to constraint propagation. For example, "Does this value have support?", or 'Is this problem generalised arc-consistent?". We identified dependencies between the tractability and intractability of these questions for finite domain variables and we have shown that these questions are intractable in general. We have then demonstrated how the same tools of computational complexity can be used in the design and analysis of specific global constraints. In particular, we have illustrated how computational complexity can be used to determine when a lesser level of local consistency should be enforced, when decomposing constraints will reduce propagation, when constraints can be combined tractably and when generalisation leads to intractability. We showed that a wide range of global constraints, both existing and new, are intractable. In particular, the `NValue` and `AtMost1` constraints, the global cardinality constraint with repeated variables and the `Common` constraint, are proven here to be intractable. We have also shown how the same tools can be used to study meta-constraints like the `Cardpath` constraint. In the future, we plan an extensive study of the computational complexity of global constraints beyond finite domain variables (e.g. on set and multiset variables).

Aknowledgements

The second and fourth author are members of the Knowledge Representation and Reasoning Programme at National ICT Australia. NICTA is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council. The third author is supported by Science Foundation Ireland. We thank Marie-Christine Lagasquie for some advice about reducibility notions.

References

[1] N. Beldiceanu. Global constraints as graph properties on a structured network of elementary constraints of the same type. Technical report, Swedish Institute of Computer Science, 2000. SICS Technical Report T2000/01.

[2] N Beldiceanu. Global constraints as graph properties on a structured network of elementary constraints of the same type. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP'00), LNCS 1894, Springer–Verlag*, pages 52–66, Singapore, 2000.

[3] N. Beldiceanu. Pruning for the minimum constraint family and for the number of distinct values constraint family. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP'01), LNCS 2239, Springer–Verlag*, pages 211–224, Singapore, 2001.

[4] N. Beldiceanu and M. Carlsson. Revisiting the cardinality operator and introducing cardinality-path constraint family. In *Proceedings ICLP'01*, pages 59–73, 2001.

[5] N. Beldiceanu and E. Contegean. Introducing global constraints in CHIP. *Mathematical Computer Modelling*, 20(12):97–123, 1994.

[6] C. Bessiere. Complexity of the `cardpath` constraint. Technical Report TR-05036, LIRMM (CNRS / University of Montpellier), Montpellier, France, January 2005.

[7] C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. Filtering algorithms for the NValue constraint. In *Proceedings CPAIOR'05*, Prague, Czech Republic, 2005.

[8] C. Bessiere and J.C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 398–404, Nagoya, Japan, 1997.

[9] C. Bessiere and P. Van Hentenryck. To be or not to be ... a global constraint. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP'03), LNCS 2833, Springer–Verlag*, pages 789–794, Kinsale, Ireland, 2003. Short paper.

[10] M. Carlsson and N. Beldiceanu. Arc-consistency for a chain of lexicographic ordering constraints. Technical report T2002-18, Swedish Institute of Computer Science, 2002. ftp://ftp.sics.se/pub/SICS-reports/Reports/SICS-T–2002-18–SE.ps.Z.

[11] Y. Caseau and F. Laburthe. Solving various weighted matching problems with constraints. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP'97), LNCS 1330, Springer–Verlag*, pages 17–31, Linz, Austria, 1997.

[12] D.A. Cohen, P. Jeavons, and M. Koubarakis. Tractable disjunctive constraints. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP'97), LNCS 1330, Springer–Verlag*, pages 478–490, Linz, Austria, 1997.

[13] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[14] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[15] Y. Deville, O. Barette, and P. Van Hentenryck. Constraint satisfaction over connected row convex constraints. In *Proceedings IJCAI'97*, pages 405–410, Nagoya, Japan, 1997.

[16] F. Focacci, A. Lodi, and M. Milano. Optimization-oriented global constraints. *Constraints*, 7:351–365, 2002.

[17] E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, Jan. 1982.

[18] A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP'02), LNCS 2470, Springer–Verlag*, Ithaca NY, 2002.

[19] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to NP-Completeness*. Freeman, San Francisco CA, 1979.

[20] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural csp decomposition methods. In *Proceedings IJCAI'99*, pages 394–399, Stockholm, Sweden, 1999.

[21] B. Hnich, Z. Kiziltan, and T. Walsh. Combining symmetry breaking with other constraints: lexicographic ordering with sums. In *Proceedings of the 8th International Symposium on the Artificial Intelligence and Mathematics*, 2004.

[22] D.E. Knuth and A. Raghunathan. The problem of compatible representatives. *SIAM Journal of Discrete Mathematics*, 5(3):422–427, 1992.

[23] O. Lhomme. Arc-consistency filtering algorithms for logical combinations of constraints. In *Proceedings CPAIOR'04*, pages 209–224, Nice, France, 2004.

[24] R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings ECAI'88*, pages 651–656, Munchen, FRG, 1988.

[25] F. Pachet and P. Roy. Automatic generation of music programs. In *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP'99), LNCS 1713, Springer–Verlag*, pages 331–345, Alexandria VA, 1999.

[26] C.H. Papadimitriou and M. Yannakakis. The complexity of facets (and some facets of complexity). *J. Comput. System Sci.*, 28:244–259, 1984.

[27] C. Quimper. Enforcing domain consistency on the extended global cardinality constraint is NP-hard. Technical Report CS-2003-39, School of Computer Science, University of Waterloo, 2003.

[28] J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings AAAI'94*, pages 362–367, Seattle WA, 1994.

[29] J.C. Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings AAAI'96*, pages 209–215, Portland OR, 1996.

[30] J.C. Régin. Cost-based arc consistency for global cardinality constraints. *Constraints*, 7:387–405, 2002.

[31] J.C. Régin and M. Rueher. A global constraint combining a sum constraint and difference constraint. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP'00), LNCS 1894, Springer–Verlag*, pages 384–395, Singapore, 2000.

[32] A. Sadler and C. Gervet. Global reasoning on sets. In *Proceedings of Workshop on Modelling and Problem Formulation (FORMUL'01)*, 2001. held alongside CP-01.

[33] M. Sellmann. Approximated consistency for knapsack constraints. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP'03), LNCS 2833, Springer–Verlag*, pages 679–693, Kinsale, Ireland, 2003.

[34] M. Sellmann. Cost-based filtering for shorter path constraints. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP'03), LNCS 2833, Springer–Verlag*, pages 694–708, Kinsale, Ireland, 2003.

[35] P. van Beek and R. Dechter. On the minimality and global consistency of row-convex constraint networks. *Journal of the ACM*, 42(3):543–561, 1995.

[36] P. Van Hentenryck and Y. Deville. The cardinality operator: a new logical connective for constraint logic programming. In *Proceedings ICLP'91*, pages 745–759, 1991.

[37] 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.

[38] M. Wallace. Practical applications of constraint programming. *Constraints*, 1:139–168, September 1996.

[39] T. Walsh. Consistency and propagation with multiset constraints: a formal viewpoint. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP'03), LNCS 2833, Springer–Verlag*, pages 724–738, Kinsale, Ireland, 2003.

[40] T. Walsh. Constraint patterns. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP'03), LNCS 2833, Springer–Verlag*, Kinsale, Ireland, 2003.