The Power of Propagation: When GAC is Enough

David A. Cohen[1] and Peter G. Jeavons[2]

[1] Department of Computer Science, Royal Holloway, University of London, UK
d.cohen@rhul.ac.uk
[2] Department of Computer Science, University of Oxford, UK
peter.jeavons@cs.ox.ac.uk

Abstract. Considerable effort in constraint programming has focused on the development of efficient propagators for individual constraints. In this paper, we consider the combined power of such propagators when applied to collections of more than one constraint. In particular we identify classes of constraint problems where such propagators can decide the existence of a solution on their own, without the need for any additional search. Sporadic examples of such classes have previously been identified, including classes based on restricting the structure of the problem, restricting the constraint types, and some hybrid examples. However, there has previously been no unifying approach which characterises all of these classes: structural, language-based and hybrid. In this paper we develop such a unifying approach and embed all the known classes into a common framework. We then use this framework to identify novel classes of problems that can be solved by propagation alone.

1 Introduction

Constraint programming (CP) is widely used to solve a variety of practical problems such as planning and scheduling [34,43], and industrial configuration [1,33]. Much of the success of CP arises from to the use of special-purpose constraint types known as *global constraints*.

Global constraints facilitate the declarative encoding of problems; they allow the constraint programmer to express high-level knowledge about relationships between variables [26, 41, 44]. A global constraint is rarely represented explicitly by listing all the assignments that satisfy it. Instead, such constraints are usually represented *implicitly* by an algorithm in the solver that decides which assignments the constraint should allow.

For many kinds of global constraints another algorithm is also provided that prunes values from the domains of variables if they can be shown to be infeasible, given the values currently available for other variables [5,42]. Such an algorithm is known as a filtering algorithm, or *propagator*.

Considerable effort in constraint programming has focused on the development of efficient propagators that can achieve various kinds of *local consistency* for individual constraints. The strongest level of local consistency that can be established for an individual constraint considered in isolation is when every value in the domain of every variable is part of an allowed assignment that assigns each variable of the constraint a value from its current domain. When this condition holds the domains are said to satisfy the property of *generalised arc-consistency* (GAC) for that constraint [5] (sometimes called *domain consistency*). An algorithm that removes values from the domains of the variables of an individual constraint to achieve this property is called a *GAC propagator* for that constraint. The close connection between GAC propagation and unit propagation in SAT-solvers is explored in [2].

Many common global constraint types, including the standard AllDifferent constraint [40], are known to have efficient GAC propagators. For an early survey of global constraints see the Handbook of Constraint Programming [34], and for a detailed description of many global constraints and associated GAC propagators see the online Global Constraint Catalog [4].

However, the development of efficient GAC propagators for *individual* constraints does not shed much light on the effectiveness of such algorithms when applied to multiple *overlapping* global constraints, which is a standard feature of most practical constraint problems.

In this paper we will consider the combined effect of running GAC propagators on each of the constraints in problems with more than one constraint. In particular, we will characterise constraint problems where using such propagators can efficiently decide whether or not a solution exists, without the need for any additional processing or search. This property will be referred to as being *decided by GAC*. The use of propagators is implemented by most existing solvers, so any such solver will be able to determine whether any instance that is decided by GAC has a solution or not, simply by using propagation.

We begin by surveying and characterizing the diverse classes of problems that have previously been shown to be decided by GAC, and then give a unified description that characterises all individual instances with this property. We then show that this characterisation provides a simple alternative explanation for each of the previously known classes.

However, we also show that we cannot expect to be able to efficiently recognise all problem instances of a certain kind that are decided by GAC, by showing that this problem is NP-hard for many kinds of problems. Finally, we give a diagram showing the relationships between the various classes.

If we can decide the existence of a solution using a certain algorithm, we can often use a simple modification of this algorithm to actually find a solution when one exists. For any class of CSP instances where we can add constant constraints (which define assignments to individual variables), we can find a solution by adding unary constant constraints on each variable in turn, restricting it to a single value, and calling the decision algorithm each time [13]. Most of the constraint problems we consider in this paper will allow arbitrary constant constraints, and in these cases for any instance that is decided by GAC we can use propagation repeatedly to find a solution when it exists.

One important application area for our results will be to find decompositions of global constraints into combinations of smaller constraints [6]. If the instance formed by the smaller constraints is decided by GAC, and retains this property when we add an arbitrary unary constant constraint, then we can enforce GAC on the original global constraint by adding unary constant constraints to each variable in turn, restricting it to a single value, and enforcing GAC on the set of smaller constraints each time.

Another application area is to identify sub-problems of a given problem that can be solved efficiently, that can be used as targets for problem reduction or pre-processing strategies [17]. We believe that the systematic identification of the properties needed for GAC decidability that we give here will lead to novel problem reduction and simplification strategies.

2 Constraints and propagators

Definition 1 (CSP instance). *A CSP instance is a triple $\langle V, D, C\rangle$, where V is a finite set of* variables*, D is a function which maps each element of V to a finite set of possible values, called its* domain*, and C is a finite set of* constraints*.*

Each constraint $c \in C$ is a pair, $\langle \sigma, \rho\rangle$, where σ is a sequence of variables from V, called the scope*. The length of σ, denoted $|\sigma|$, is called the arity of c. The* relation*, ρ, is a subset of $D(\sigma[1]) \times \cdots \times D(\sigma[r])$, where $r = |\sigma|$, and defines the allowed combinations of values for the list of variables in σ.*

A solution *to a CSP instance is a function which maps each variable to a value from its domain in such a way that all constraints are satisfied.*

CSP instances are abstract specifications of problems: they tell us what the required properties of the instance are, but do not tell us how that instance should be represented for processing by a constraint solver. As discussed in [11], when the constraints in a family of problems have unbounded arity, the way that the constraints are *represented* can significantly affect their complexity.

In this paper we will assume that the constraints in our instances are represented by pre-defined global constraints that impose the specified restrictions, each with an associated GAC propagator that the solver can use to prune values from the domains of the variables in the scope of that constraint.

A standard approach to processing a CSP instance, implemented in many current solvers, is to run the GAC propagators on each constraint until no further changes result. If doing this removes all possible values from the domain of at least one variable, then we will say that this algorithm returns the answer "no". This outcome will be called "domain wipeout". Any other outcome (i.e., at least one remaining value in the domain of every variable) corresponds to returning the answer "yes".

Running this algorithm on any instance that has a solution will always return the value "yes", but running it on instances with no solutions may also in some cases return the value "yes". Such cases will need additional processing to determine whether a solution actually exists (such as some form of search).

We will say that an individual CSP instance is *decided by GAC* if running this algorithm returns the answer "yes" if the instance has a solution, and returns the answer "no" if it does not. This is captured by the following definition.

Definition 2. *A CSP instance is* decided by GAC *if it has a solution, or else repeatedly running GAC propagators on each of its separate constraints leads to domain wipeout.*

As the next examples illustrate, it can be challenging to distinguish between instances where GAC decides and instances where it does not.

Example 1. A Latin square is an arrangement of the numbers 1 to n in an $n \times n$ square grid in such a way that the numbers in each row are distinct and the numbers in each column are distinct. The task of completing a Latin square where some entries are already given and others are left blank is sometimes referred to as the *quasi-group completion* problem [39] and has been used as a benchmark problem for constraint programming.

It can be formulated as a constraint problem where we have n^2 variables, some with a single specified value, and others with domain values 1 to n, and AllDifferent constraints on the rows and columns. Empirical studies of this formulation have shown that in many cases, especially when n is small, GAC propagation alone will decide whether a given instance of this problem has a solution without the need for any further search [27]. However, this is not true in general, as this problem is known to be NP-complete [16].

1	2			
2	1			
		3		
			3	
				3

1	3	2	4	5
2	1			
3		1		
4			1	
5				1

Fig. 1. Two partial Latin squares with no solution

Two instances of such a problem are shown in Figure 1. All the empty squares have two or more possible values that are distinct from the already-assigned values in the same row and column. However, using the formulation above the instance on the left is decided by GAC because the GAC propagators will remove all of the remaining values from the domains of the variables. The instance on the right is not decided by GAC as propagation removes no further values.

Example 2. Consider the CSP instance I_{TET} which has variables $\{v_1, v_2, v_3, v_4\}$ each with domain $\{R, G, B\}$ and four ternary AllDifferent constraints, with scopes $\langle v_1, v_2, v_3 \rangle, \langle v_1, v_2, v_4 \rangle, \langle v_1, v_3, v_4 \rangle, \langle v_2, v_3, v_4 \rangle$, and a unary constraint on variable v_4 that allows only the single value R.

This instance has no solution. Running a GAC propagator on the unary constraint reduces the domain of v_4 to the single value R. Then running a GAC propagator on the constraint with scope $\langle v_2, v_3, v_4\rangle$ will remove the value R from the domains of v_2 and v_3. Then running a GAC propagator on the constraint with scope $\langle v_1, v_2, v_3\rangle$ will remove the values B and G from the domain of v_1. Finally, running a GAC propagator on the constraint with scope $\langle v_1, v_2, v_4\rangle$ will remove the value R from the domain of v_1, causing a domain wipeout.

Hence I_{TET} is decided by GAC.

The following example shows that removing a constraint from an instance will, in some cases, stop GAC deciding that instance.

Example 3. Now consider the CSP instance I'_{TET} which has the same variables and domains as the instance I_{TET} in Example 3, but without the unary constraint. This instance again has no solution, but it is now generalised arc consistent, so running a GAC propagator on each constraint has no effect.

Hence I'_{TET} is *not* decided by GAC.

3 Restricted classes decided by GAC

In this section we survey the classes already known to be decided by GAC.

3.1 Structural restrictions

The first kind of restriction that we consider is to limit the way that the constraints in a given instance share their variables, or, in other words, the way that the constraint scopes overlap [14, 28, 31]. If every instance in a class defined by a structural restriction is decided by GAC it means that we can apply arbitrary constraints over the same scopes and the result will still be decided by GAC.

It is well-known that any *binary* CSP instance where the constraint scopes form a tree is decided by GAC [24]. To obtain a simple generalisation of this result to non-binary CSP instances we need to identify a suitable generalisation of the notion of a tree.

One possible generalisation of the graph-theoretic notion of a tree that has received a great deal of attention is the class of *acyclic hypergraphs* [3]. A hypergraph is a generalisation of the idea of a graph, where the edges can contain an arbitrary number of vertices, rather than just two (the edges in such a structure are sometimes referred to as hyperedges). A hypergraph is said to be acyclic if repeatedly removing all hyperedges contained in other hyperedges, and all vertices contained in only a single hyperedge, eventually deletes all vertices.

Another class of hypergraphs that has been considered in this context are those with *bounded tree-width* [19].

Both acyclicity and bounded tree-width have proven very useful in the analysis of the computational complexity of many combinatorial search problems. Indeed, many NP-hard problems become tractable if their structure is acyclic or has bounded tree-width.

Solving a CSP instance whose constraints are represen[illegible]e., as table constraints) is known to be tractable if the hypergraph defined by the constraint scopes is acyclic [32]. However, this is no longer true if the constraints are represented implicitly, even when they have a fixed, finite domain [15].

check into this.

Dalmau et al. [19], building on several earlier results [21, 25], showed that the class of all CSP instances whose associated hypergraphs belong to some (recursively enumerable) family with bounded tree-width is solvable in polynomial time. This remains true even when the constraints are represented implicitly.

However, restricting the structure of CSP instances to be acyclic or to have bounded tree-width does not ensure that they are decided by GAC. Moreover, there are structural classes of CSP instances that are decided by GAC but do not have bounded tree-width (for example, the class of instances containing a single constraint of unbounded arity). Hence we need a different generalisation of trees in order to be able to characterise the structural classes of CSP instances that are decided by GAC.

Definition 3. *A variable v is called an* articulation point *for a set of constraints if those constraints can be partitioned into two non-empty sets whose scopes share only the variable v.*

A CSP instance is Berge-acyclic *[3] if every variable is either an articulation point or belongs to at most one constraint scope.*

It is clearly the case that every *binary* CSP instance where the constraint scopes form a tree, is Berge-acyclic. However, there is no requirement for a Berge-acyclic instance to be connected, so any binary CSP instance where the constraint scopes form a forest is also Berge-acyclic.

It has been noted by many authors that Freuder's result about trees can be extended to non-binary Berge-acyclic instances. Here we state a slightly stronger result: these are the *only* structural classes which are decided by GAC.

Theorem 1. *The following are equivalent:*

- *The CSP instance I is Berge-acyclic;*
- *Every CSP instance with the same constraint scopes as I is decided by GAC.*

Proof. If I is Berge-acyclic, then the constraint scopes only overlap at articulation points. Hence, after establishing GAC, if the domains are not empty, then for each connected subset of constraints we can choose any constraint as the root, choose any allowed tuple for that constraint, extend the assigned values to allowed values for the children, and repeat until we reach the leaves. Hence I has a solution, and so is decided by GAC.

If I is not Berge-acyclic, then there exists a sequence of two or more distinct variables each shared by two or more constraint scopes where each successive pair are contained in the scope of some constraint, and the first and last are also in the scope of some constraint. So, choose variables $x_1, x_2, \ldots, x_k$ and scopes $\sigma_i, 1 \leq i \leq k$ such that $\{x_i, x_{i+1}\} \subseteq \sigma_i$ for $i = 1, \ldots, k-1$ and $\{x_k, x_1\} \subseteq \sigma_k$. On σ_1 apply a constraint that requires $x_i \neq x_{i+1}$, and on all other such scopes σ_i apply a constraint that requires $x_i = x_{i+1}$. If no further restrictions are imposed, then the resulting instance is GAC but has no solution.

3.2 Language restrictions

The second kind of restriction that we consider is a restriction on the constraint relations that can be specified for the constraints in a given instance, or, in other words, the kinds of constraints we can use [8, 35].

It is convenient to refer to a set of relations over some fixed set D as a *constraint language*, and to refer to a class of CSP instances where the constraint relations of all constraints are elements of Γ as the class of CSP instances over the language Γ.

Amongst the earliest such language restrictions to be identified were the so-called *min-closed* and *max-closed* families of constraints [36]. These constraint types generalise Horn clauses to larger domains, and also generalise the basic arithmetic constraints provided in the CHIP programming language [36]. Any class of CSP instances where the constraints are all max-closed or all min-closed is decided by GAC [29]. This result generalises the well-known fact that unit propagation decides all satisfiability problems over Horn clauses.

This class of constraints was further generalised to the class of all constraints where the constraint relations are preserved by a so-called semi-lattice polymorphism [8]. Another generalisation has been described [20], to constraints where the constraint relations are preserved by a set function. A set function on a set D is a function from the non-empty subsets of D to D.

Definition 4. *A relation ρ of arity r is said to be preserved by a set function f if, for any non-empty subset $\{t_1, t_2, \ldots, t_k\}$ of tuples from ρ, the tuple*

$$\langle f(\{t_1[1], t_2[1], \ldots, t_k[1]\}), \ldots, f(\{t_1[r], t_2[r], \ldots, t_k[r]\})\rangle$$

is again an element of ρ. A language Γ is said to be preserved by a set function if every relation in Γ is preserved by that set function.

Building on the work of [22], Dalmau and Pearson were able to show that any CSP instance over a fixed domain where the constraint relations are preserved by a set function is decided by GAC [20]. In fact they obtained the following result, for which we include a short proof in our terminology, so that we can extend this result below.

Theorem 2 ([20]). *The following are equivalent:*

- *A constraint language Γ over a finite set D is preserved by a set function;*
- *Every CSP instance over Γ is decided by GAC.*

Proof. Let Γ be a constraint language over a finite set D. We construct a canonical CSP instance I_D over Γ, as follows.

The variables of I_D are the non-empty subsets of D. For each relation $\gamma \in \Gamma$, with arity r we impose the constraint $\langle (A_1, \ldots, A_r), \gamma\rangle$ (where the A_i are not necessarily distinct), for all choices of $(A_1, \ldots, A_r)$ that satisfy the following condition: for every $1 \leq i \leq r$ and every $a_i \in A_i$ there exist elements a_j in each of the remaining A_j for which $\gamma(a_1, \ldots, a_r)$ holds. Observe that the solutions to I_D are precisely the set functions that preserve Γ.

Assume first that GAC decides every CSP instance over Γ, so GAC decides I_D. Restricting the domain of each variable A_i to be the set A_i gives a sub-instance of I_D with non-empty domains which is GAC. Hence, by our assumption, I_D has a solution and so, by the observation above, every relation in Γ is preserved by the set function that corresponds to this solution.

Conversely suppose that not every instance over Γ is decided by GAC. In this case there is some instance I over Γ with non-empty domains which is GAC but has no solution. Let D' be the mapping from the variables to their sub-domains after enforcing GAC. By our construction of I_D, the mapping which maps each variable v of I to the variable $D'(v)$ in I_D gives a mapping from I to I_D that maps each constraint scope of I to a list of variables in I_D that are constrained in the same way. Hence I_D is also not solvable, and so, by the observation above, Γ is not preserved by any set function.

Note that if all the constraint relations in some CSP instance are preserved by a set function then we can remove any constraint and still have this property. However, as we have seen in Examples 2 and 3, not every instance that is decided by GAC is still decided by GAC after removing a constraint. Hence not every individual instance that is decided by GAC will have all its constraint relations preserved by some set function; rather, as Theorem 2 indicates, this will only be the case for those instances where all other instances over the same constraint language are also decided by GAC, which is quite a strong requirement.

3.3 Hybrid restrictions

The third kind of restriction that we consider is restriction on both the scopes and the constraint relations that can be specified for the constraints in a given instance [12, 18, 37].

Amongst the earliest such hybrid restrictions to be identified were the so-called *triangulated* CSP instances described in [12]. These instances contain only binary constraints, so for any instance $\langle V, D, C\rangle$ there is an associated graph with set of vertices $V \times D$, and edges between each pair of distinct values for the same variable, and each pair of values for distinct variables that is forbidden by a constraint. Such a graph is called the *microstucture complement* of the instance. It is known [12] that any instance where this graph is triangulated is decided by GAC. It has also been shown that the class of all such instances is not defined by any structural restriction, nor by any language restriction.

More recently, another hybrid restriction defining a class of binary CSP instances that are decided by GAC has been identified [18]. These are the instances satisfying the so-called *broken-triangle property.*

Definition 5. *A binary CSP instance satisfies the* broken-triangle property *(BTP) with respect to the variable ordering* $<$*, if there is at most one constraint on each pair of variables, and for all triples of variables* v_i, v_j, v_k *such that* $v_i < v_j < v_k$*, if*

- *the pair of values* $\langle a, b\rangle$ *is allowed on the variables* $\langle v_i, v_j\rangle$*; and*

- *the pair of values* $\langle a, c\rangle$ *is allowed on the variables* $\langle v_i, v_k\rangle$*; and*
- *the pair of values* $\langle b, d\rangle$ *is allowed on the variables* $\langle v_j, v_k\rangle$*;*

then either

- *the pair of values* $\langle a, d\rangle$ *is allowed on the variables* $\langle v_i, v_k\rangle$*; or*
- *the pair of values* $\langle b, c\rangle$ *is allowed on the variables* $\langle v_j, v_k\rangle$*.*

CSP instances satisfying the BTP (or its various extensions [17]) are the only known examples of classes of instances decided by GAC which are not closed under the action of removing a constraint, as the next example illustrates.

Example 4. Consider a CSP instance I with variables v_1, v_2, v_3, v_4 each with domain $\{0, 1, 2\}$ and constraints $v_1 < v_2$, $v_2 < v_3$, $v_3 < v_4$, and $v_1 \geq v_4$, together with an additional constraint on v_1 and v_3 that allows only the combinations $\{(2, 0), (2, 1), (2, 2), (1, 2)\}$.

This instance satisfies the BTP, and so is decided by GAC. However, if we remove the constraint on v_1 and v_3 to obtain a reduced instance I' it no longer satisfies the BTP (consider the triple of variables v_1, v_3, v_4). On the other hand, the reduced instance I' is now max-closed, so is still decided by GAC.

4 A characterisation of instances decided by GAC

To unify the earlier results and obtain more fine-grained results that characterise all individual instances decided by GAC, we need to consider both the structure of the instance (defined by the constraint scopes) and the language of the instance (defined by the constraint relations). To do this we will treat each constraint $\langle \sigma, \rho\rangle$ as a "labelled relation" where each component of the relation ρ is labelled by the corresponding entry (variable) in the scope σ. We then consider what other instances can be formed using these labelled relations. We allow the constraints in these new instances to share a variable only when they share the same label for the corresponding component. (Note that this approach is very similar to the machinery developed in [9] for multi-sorted constraint relations.)

Definition 6. *Given any CSP instance* $I = \langle V, D, C\rangle$*, we say that an instance* $I' = \langle V', D', C'\rangle$ *is an instance* over the same labelled language *as* I*, if there is a mapping* λ *from* V' *to* V*, called a* labelling*, such that for every constraint* $c' = \langle (v'_1, \ldots, v'_r), \rho'\rangle \in C'$*, the constraint* $\langle (\lambda(v'_1), \ldots, \lambda(v'_r)), \rho'\rangle$ *is an element of* C *and for all* $v' \in V'$*, the domain* $D'(v') = D(\lambda(v'))$ *.*

Lemma 1. *The following are equivalent:*

- *Applying GAC propagators to the CSP instance* I *leads to domain wipeout;*
- *There is some Berge-acyclic instance over the same labelled language as* I *which has no solution.*

Proof. Let $I = \langle V, D, C\rangle$ be a CSP instance, and let $\mathrm{GAC}(c, v, d)$ mean that a GAC propagator applied to constraint $c \in C$ deletes value $d \in D(v)$ from the domain of $v \in V$.

First, suppose that GAC applied to the instance I leads to domain wipeout at v. We must have a sequence: $\mathrm{GAC}(c_1, v_1, d_1), \ldots, \mathrm{GAC}(c_m, v_m, d_m)$ in which every value originally in the domain of $v = v_m$ is deleted at some point.

Now we use this sequence to inductively build a Berge-acyclic instance over the same labelled language as I which will have no solution.

We begin the construction with an empty instance. Assume that for each $j < k$ we have constructed a Berge-acyclic instance for $\mathrm{GAC}(c_j, v_j, d_j)$. We can then build the instance for $\mathrm{GAC}(c_k, v_k, d_k)$ as follows. Let $c_k = \langle \sigma_k, \rho_k \rangle$, where $\sigma_k = (v_{i_1}, \ldots, v_{i_r})$. Create variables $v^k_{i_1}, \ldots, v^k_{i_r}$ with labels $v_{i_1}, \ldots, v_{i_r}$ and add the constraint $\langle (v^k_{i_1}, \ldots, v^k_{i_r}), \rho_k \rangle$. Now, for each $j < k$ and each $\mathrm{GAC}(c_j, v_j, d_j)$ where $v_j \in \sigma(k)$ we add (a separate copy of) the Berge-acyclic instance constructed for $\mathrm{GAC}(c_j, v_j, d_j)$ and identify the variables v^j_j and v^k_j.

All the variables that we identify during this construction are articulation points, so the resulting instance is Berge-acyclic. Moreover, since every constraint has the same relation as some constraint in C, and constraint scopes only overlap when the variables have the same label, the constructed instance is over the same labelled language as the original instance.

To see that it has no solution, it is enough to observe that at stage k a variable v^k cannot take any value d for which there is some $i \leq k$ and deletion $\mathrm{GAC}(c_i, v_i, d_i)$ where $v_i = v$ and $d_i = d$. hence, by our assumption about the sequence of GAC applications, there are no possible values for the variable v_m at stage m.

Conversely, suppose that some Berge-acyclic instance $T = \langle V', D', C' \rangle$ over the same labelled language as I has no solution. Since Berge-acyclic instances are decided by GAC, by Theorem 1, we know that applying GAC propagators to the constraints of T in some order removes all values from some domain.

Since T is Berge-acyclic, the constraints of T can be arranged in a forest whose edges correspond to the articulation points. Choose the tree in this forest where the domain wipeout occurs, and choose the constraint whose propagator removes the final value from the domain as the root. Order the constraints so that each parent occurs after all of its descendants (i.e., choose a post-order on the tree). Now we know that applying GAC propagators to the constraints in T along this post-order from each leaf to the root leads to domain wipeout at the root.

Hence we have a sequence: $\mathrm{GAC}(c'_1, v'_1, d_1), \ldots, \mathrm{GAC}(c'_m, v'_m, d_m)$ for T, in which every value originally in the domain of v'_m is deleted at some point. By Definition 6, each variable v'_i in this sequence corresponds to a variable $\lambda(v'_i)$ in the original instance I, and each constraint c'_i corresponds to a constraint in the original instance I, which we will call $\lambda(c'_i)$. If we apply the GAC propagators to each of the corresponding constraints $\lambda(c'_i)$ of I in the same order, we claim that after each application the domain of $\lambda(v'_i)$ will be a subset of the domain of v'_i at the same point in the process.

We will establish this claim by induction. It is clearly true at the start of the process because both variables start with the same domain, by Definition 6. Suppose that it is true for the first $(k-1)$ applications in the sequence. Consider the next application of the GAC propagator, to c'_k. By our hypothesis, the domains of all variables in I corresponding to children of c'_k in T are subsets of the domains of the corresponding variables in T. Hence applying the GAC propagator on constraint $\lambda(c'_k)$ removes at least as many values from the domain of $\lambda(v'_k)$ as its analogue removes from v'_k, so the claim follows by induction.

It follows that this sequence of applications of GAC propagators to the constraints in I leads to domain wipeout at $\lambda(v'_m)$, which proves the result.

Theorem 3. *The following are equivalent:*

- *An instance I is decided by GAC;*
- *I has a solution if and only if every Berge-acyclic instance over the same labelled language as I has a solution;*

Proof. First, suppose that the CSP instance I is decided by GAC.

If I has a solution, say s, then we can use s to solve any Berge-acyclic instance T over the same labelled language as I. Simply choose the value of each variable in T to be $s(\lambda(v))$.

If I has no solution, since GAC decides I we know that GAC leads to domain wipeout and we can appeal to Lemma 1 to obtain a Berge-acyclic instance over the same labelled language as I which has no solution.

Conversely, suppose that I satisfies the second condition in the statement. If I has a solution then GAC decides, since GAC preserves solutions. On the other hand, if I has no solution then, by our assumption, some Berge-acyclic instance T over the same labelled language as I has no solution, and again we can appeal to Lemma 1 to show that applying GAC propagators to I leads to domain wipeout.

A similar property was identified in [22], where it is referred to as "tree duality", but it was only defined for classes of instances over a fixed constraint language (and was expressed rather more abstractly, in terms of Datalog and algebraic conditions).

Theorem 3 does not bound the size of the Berge-acyclic instances that need to be considered. However, examining the proof of Lemma 1, we can see that the only Berge-acyclic instances we need to consider correspond to sequences of domain reductions caused by GAC propagators. Since the maximum number of domain reductions that can occur in a CSP instance $\langle V, D, C\rangle$ is $\sum_{v\in V} |D(v)|$, it follows that it is sufficient to consider only Berge-acyclic instances of depth at most $\sum_{v\in V} |D(v)|$.

In fact, for any CSP instance I we can identify a single "universal" Berge-acyclic instance I_B over the same labelled language as I (see Algorithm 1).

Corollary 1. *The following are equivalent:*

- *An instance I is decided by GAC;*
- *I has a solution if and only if I_B has a solution.*

It follows that the second property holds for all of the classes decided by GAC that we have described in earlier sections. For Berge-acyclic instances this follows immediately by the following observation.

Algorithm 1 Building $I_B = \langle V', D', C' \rangle$ from instance $I = \langle V, D, C \rangle$

```
Set $V' = \emptyset$;  $C' = \emptyset$
for all connected components $K \subseteq C$ do
    Call MakeTree($v, -, 1$) for some $v$ that occurs in the scopes of $K$
end for
Set $D'$ so that $D'(v') = D(v)$ for all $v' \in V$, where $v$ is the label of $v'$

function MakeTree($v, con, depth$)
    Add a new variable $v'$ to $V'$ with label $v$
    if $depth \leq \sum_{v \in V} |D(v)|$ then
        for all constraints $c = \langle \sigma, \rho \rangle$ of $I$ except $con$ do
            if $v$ occurs in $\sigma$ then
                for all $v_i \in \sigma$ except $v$ do
                    $v'_i$ = MakeTree($v_i, c, depth + 1$)
                end for
                Add a constraint on $v'$ and all the $v'_i$, with relation $\rho$, to $C'$
            end if
        end for
    end if
    return $v'$
end function
```

Observation 1 *If I is Berge-acyclic, then $I_B = I$.*

Theorem 3 also gives an alternative, more illuminating, proof that every instance whose language is preserved by a set function is decided by GAC.

Proposition 1. *Any instance I whose constraint relations are preserved by a set function has a solution if and only if every Berge-acyclic instance over the same labelled language as I has a solution;*

Proof. Let Γ_I be the set of constraint relations of an instance I, and assume that Γ_I is preserved by a set function.

If I has a solution s, then every Berge-acyclic instance over the same labelled language as I has a solution, given by applying s to the label of each variable.

Conversely, assume that every Berge-acyclic instance over the same labelled language as I has a solution. In this case, by Lemma 1, establishing GAC on I cannot lead to domain wipeout. Hence, after establishing GAC, every variable of I has a non-empty domain. Now apply the set function to these domains and we get a value at each variable that satisfies all the constraints, and hence a solution to I.

Another simple consequence of Theorem 3 is that whenever an instance I can be shown to have a solution just by showing that some particular Berge-acyclic instance over the same labelled language has a solution, then I is decided by GAC.

We will now show that for any instance I that satisfies the BTP (Definition 5), I has a solution if a particular Berge-acyclic instance over the same labelled language, which we will call I_B^{ord}, has a solution. By the observation just made, this will be enough to show that any instance satisfying the BTP is decided by GAC.

To obtain I_B^{ord} we take into account the ordering on the variables along which they satisfy the BTP, and we replace the "occurs" check in Algorithm 1 by a test to check whether v is the *earliest* variable in the relevant scopes according to this ordering. For instances I that are not Berge-acyclic, the instance I_B^{ord} can be much smaller than I_B, as the following example illustrates.

Example 5. The scopes of the instance I_{TET} described in Example 3 are shown in Figure 2, along with the scopes of the corresponding instance I_B^{ord} for the ordering $v_1 < v_2 < v_3 < v_4$.

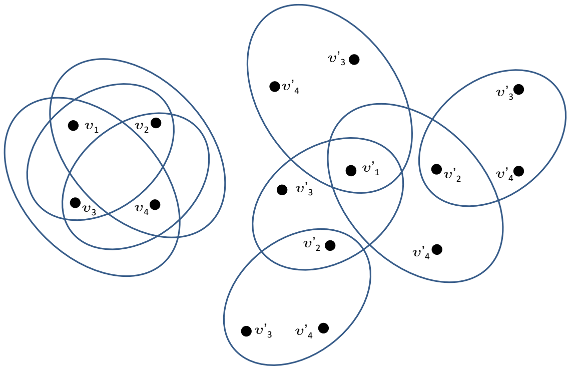

Fig. 2. The instance I described in Example 3 and the corresponding instance I_B^{ord}

Theorem 4. *A BTP instance I has a solution if I_B^{ord} has a solution.*

Proof. Let I be a binary CSP instance that satisfies the BTP with respect to the variable ordering $v_1 < v_2 < \cdots < v_n$, and assume that I_B^{ord} has a solution. We will prove by induction on the number of variables in I that I has a solution.

The inductive hypothesis is that, for every instance I with at most k variables, if I satisfies the BTP, and there is a solution s to I_B^{ord}, then there is a solution s_λ to I_B^{ord}, that assigns values to variables dependent only on their label. The solution s_λ will clearly induce a solution on I.

We begin the induction by observing that the result is trivially true for instances satisfying the BTP which have at most two variables, since they can have at most one constraint, so the instances I and I_B^{ord} are identical.

For larger instances, let I' be the instance obtained from I by removing variable v_n and all constraints involving variable v_n. It follows directly from the definition of the BTP (Definition 5) that I' also satisfies the BTP.

The solution s to I_B^{ord}, restricted to variables with labels $v_1, v_2, \ldots, v_{n-1}$, gives a solution s' to $I_B'^{ord}$. Hence, by our inductive hypothesis, we can assume that the value of s' only depends on the label of its argument.

By construction, each variable with label v_n in I_B^{ord} is constrained by just one constraint. Assigning all variables with earlier labels according to s' restricts the domain of each of these variables with label v_n to some subset of their domain. It was shown in [18, Lemma 2.3] that these subsets are totally ordered by inclusion. Moreover, since I_B^{ord} has a solution, none of them are empty. Hence they all have a common element, which can be used to extend s' to a solution s'' for I_B^{ord} where the value assigned to each variable depends only on that variable's label.

Corollary 2. *Every BTP instance is decided by GAC.*

A generalisation of the notion of BTP to non-binary constraints is described in [17] and referred to as the DGABTP. The proof of Theorem 4 can be extended to this more general class, which shows that it is also decided by GAC. Hence this more general class is tractable when the constraints are represented by GAC propagators, which extends the claims made in [17]. As this is a new result we present it as a theorem.

Theorem 5. *Every instance in the class DGABTP [17] is decided by GAC and hence is tractable when constraints are represented by GAC propagators.*

5 The complexity of identifying classes decided by GAC

We describe a class of CSP instances as NP-hard if it is NP-hard to decide whether a given instance has a solution.

Theorem 6. *Let Φ be an NP-hard class of CSP instances, where each constraint has a polynomial-time GAC propagator. It is NP-hard to determine whether a given instance from Φ is decided by GAC.*

Proof. We will show that deciding whether an instance I of Φ has a solution can be reduced, in polynomial time, to the problem of determining whether it is decided by GAC, and hence is NP-hard.

Assume we have an algorithm to determine whether I is decided by GAC.

If this algorithm return "no" for instance I, then I has no solution, since all instances with a solution are decided by GAC.

Otherwise the algorithm returns "yes" and I is decided by GAC. We can establish GAC in polynomial time by running each of the propagators on the constraints until no further changes result. If there is a domain wipeout we can conclude that I has no solution, otherwise we conclude that I has a solution.

The restricted case of 2-valued CSP instances includes the 3-SAT problem, which has polynomial-time GAC propagators but is NP-hard. Hence Theorem 6 can be applied to the class of 2-valued instances. An exactly analogous argument using the 3-colouring problem shows that it also applies to binary CSP instances.

However, for all of the specific sub-classes decided by GAC described earlier, membership can be determined in polynomial time. The inclusion relationships between these classes are shown in Figure 3; shading indicates that instances in the class are tractable, and a dashed border indicates that membership in a class is NP-hard to determine. Note that all instances with a tree structure satisfy the BTP [18].

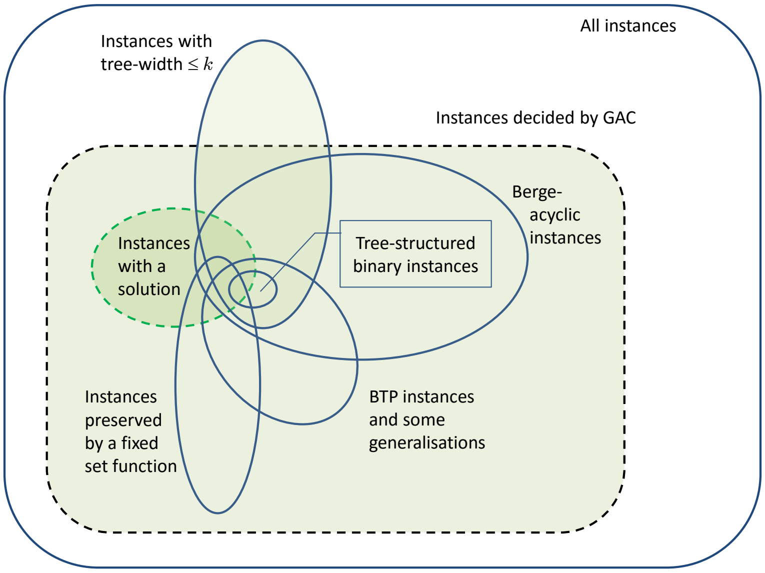

Fig. 3. The relationships between the classes of CSP instances discussed in this paper

6 Summary and related work

We have described a new characterisation for the class of CSP instances which are decided by establishing generalised arc-consistency. Our results unify and generalize several previously studied classes of problems, including tree-structured problems [24], problems with max-closed constraints [36], problems where the constraints are preserved by a set function [20], and problems with the broken-triangle property [18].

There has been a long series of earlier papers attempting to identify tractable constraint problems [8, 14, 28, 31, 38]. However, much of this previous theoretical work has assumed (often tacitly) that the constraints are represented *explicitly*, by a table of allowed assignments, and so can be modified and combined efficiently. Hence very few of these earlier theoretical results are directly applicable to overlapping constraints represented by propagators. This may be one reason why such work has had little practical impact on the design of constraint solvers.

Exceptions include the pioneering work of Bulatov and Marx [10], the structural classes explored in [30], some work on overlapping AllDifferent constraints [7, 23], and our earlier work on global constraints with a high degree of symmetry [15].

We see this paper as another step in the development of a more robust and applicable theory of complexity for realistic constraint problems which involve overlapping global constraints represented by propagators.

References

1. Aschinger, M., Drescher, C., Friedrich, G., Gottlob, G., Jeavons, P., Ryabokon, A., Thorstensen, E.: Optimization methods for the partner units problem. In: Proceedings of (CPAIOR'11). Lecture Notes in Computer Science, vol. 6697, pp. 4–19. Springer (2011)
2. Bacchus, F.: GAC via unit propagation. In: Principles and Practice of Constraint Programming - CP 2007. Lecture Notes in Computer Science, vol. 4741, pp. 133–147. Springer (2007)
3. Beeri, C., Fagin, R., Maier, D., Yannakakis, M.: On the desirability of acyclic database schemes. Journal of the ACM 30, 479–513 (1983)
4. Beldiceanu, N., Carlsson, M., Demassey, S., Petit, T.: Global constraint catalogue: Past, present and future. Constraints 12(1), 21–62 (2007)
5. Bessiere, C., Hebrard, E., Hnich, B., Walsh, T.: The complexity of reasoning with global constraints. Constraints 12, 239–259 (2007)
6. Bessiere, C., Katsirelos, G., Narodytska, N., Quimper, C., Walsh, T.: Decompositions of all different, global cardinality and related constraints. In: IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence. pp. 419–424 (2009)
7. Bessiere, C., Katsirelos, G., Narodytska, N., Quimper, C., Walsh, T.: Propagating conjunctions of alldifferent constraints. In: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010 (2010)
8. Bulatov, A., Jeavons, P., Krokhin, A.: Classifying the complexity of constraints using finite algebras. SIAM Journal on Computing 34, 720–742 (2005)
9. Bulatov, A.A., Jeavons, P.: An algebraic approach to multi-sorted constraints. In: Principles and Practice of Constraint Programming - CP 2003. Lecture Notes in Computer Science, vol. 2833, pp. 183–198. Springer (2003)
10. Bulatov, A.A., Marx, D.: The complexity of global cardinality constraints. Logical Methods in Computer Science 6, 1–27 (2010)
11. Chen, H., Grohe, M.: Constraint satisfaction with succinctly specified relations. Journal of Computer and System Sciences 76, 847–860 (2010)
12. Cohen, D.A.: A new classs of binary csps for which arc-constistency is a decision procedure. In: Principles and Practice of Constraint Programming - CP 2003. Lecture Notes in Computer Science, vol. 2833, pp. 807–811. Springer (2003)
13. Cohen, D.A.: Tractable decision for a constraint language implies tractable search. Constraints 9, 219–229 (2004)
14. Cohen, D.A., Jeavons, P., Gyssens, M.: A unified theory of structural tractability for constraint satisfaction problems. Journal of Computer and System Sciences 74(5), 721–743 (2008)
15. Cohen, D.A., Jeavons, P.G., Thorstensen, E., Zivny, S.: Tractable combinations of global constraints. In: Principles and Practice of Constraint Programming CP 2013. Lecture Notes in Computer Science, vol. 8124, pp. 230–246. Springer (2013)
16. Colbourn, C.J.: The complexity of completing partial Latin squares. Discrete Applied Mathematics 8, 25–30 (1984)
17. Cooper, M.C., Duchein, A., Mouelhi, A.E., Escamocher, G., Terrioux, C., Zanuttini, B.: Broken triangles: From value merging to a tractable class of general-arity constraint satisfaction problems. Artificial Intelligence 234, 196–218 (2016)
18. Cooper, M.C., Jeavons, P.G., Salamon, A.Z.: Generalizing constraint satisfaction on trees: Hybrid tractability and variable elimination. Artificial Intelligence 174(9-10), 570–584 (2010)

19. Dalmau, V., Kolaitis, P.G., Vardi, M.Y.: Constraint satisfaction, bounded treewidth, and finite-variable logics. In: Principles and Practice of Constraint Programming (CP'02). Lecture Notes in Computer Science, vol. 2470, pp. 223–254. Springer (2002)
20. Dalmau, V., Pearson, J.: Closure functions and width 1 problems. In: Principles and Practice of Constraint Programming - CP'99. Lecture Notes in Computer Science, vol. 1713, pp. 159–173. Springer (1999)
21. Dechter, R., Pearl, J.: Tree clustering for constraint networks. Artificial Intelligence 38, 353–366 (1989)
22. Feder, T., Vardi, M.Y.: The computational structure of monotone monadic SNP and constraint satisfaction: A study through datalog and group theory. SIAM Journal on Computing 28, 57–104 (1998)
23. Fellows, M.R., Friedrich, T., Hermelin, D., Narodytska, N., Rosamond, F.A.: Constraint satisfaction problems: Convexity makes alldifferent constraints tractable. Theoretical Computer Science 472, 81–89 (2013)
24. Freuder, E.C.: A sufficient condition for backtrack-free search. Journal of the ACM 29, 24–32 (1982)
25. Freuder, E.C.: Complexity of k-tree structured constraint satisfaction problems. In: Proceedings of the 8th National Conference on Artificial Intelligence. pp. 4–9. AAAI Press / The MIT Press (1990)
26. Gent, I.P., Jefferson, C., Miguel, I.: MINION: A fast, scalable constraint solver. In: Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06). pp. 98–102. IOS Press (2006), http://dl.acm.org/citation.cfm?id=1567016.1567043
27. Gomes, C.P., Shmoys, D.B.: The promise of LP to boost CSP techniques for combinatorial problems. Proceedings (CP-AI-OR02) pp. 25–27 (2002)
28. Gottlob, G., Leone, N., Scarcello, F.: A comparison of structural CSP decomposition methods. Artificial Intelligence 124, 2000 (2000)
29. Green, M.J., Cohen, D.A.: Tractability by approximating constraint languages. In: Principles and Practice of Constraint Programming - CP 2003. Lecture Notes in Computer Science, vol. 2833, pp. 392–406. Springer (2003)
30. Green, M.J., Jefferson, C.: Structural tractability of propagated constraints. In: Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming (CP'08). Lecture Notes in Computer Science, vol. 5202, pp. 372–386. Springer (2008)
31. Grohe, M.: The complexity of homomorphism and constraint satisfaction problems seen from the other side. Journal of the ACM 54, 1–24 (2007)
32. Gyssens, M., Jeavons, P.G., Cohen, D.A.: Decomposing constraint satisfaction problems using database techniques. Artificial Intelligence 66, 57–89 (1994)
33. Hermenier, F., Demassey, S., Lorca, X.: Bin repacking scheduling in virtualized datacenters. In: Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP'11). Lecture Notes in Computer Science, vol. 6876, pp. 27–41. Springer (2011)
34. van Hoeve, W.J., Katriel, I.: Global constraints. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, Foundations of Artificial Intelligence, vol. 2, chap. 6, pp. 169–208. Elsevier (2006)
35. Jeavons, P., Cohen, D.A., Gyssens, M.: Closure properties of constraints. Journal of the ACM 44, 527–548 (1997)
36. Jeavons, P., Cooper, M.C.: Tractable constraints on ordered domains. Artificial Intelligence 79, 327–339 (1995)

37. Kumar, T.K.S.: A framework for hybrid tractability results in boolean weighted constraint satisfaction problems. In: Principles and Practice of Constraint Programming CP 2008. Lecture Notes in Computer Science, vol. 5202, pp. 282–297. Springer (2008)
38. Marx, D.: Tractable hypergraph properties for constraint satisfaction and conjunctive queries. In: Proceedings of the 42nd ACM Symposium on Theory of Computing, (STOC'10). pp. 735–744. ACM (2010)
39. Pesant, G.: CSPLib problem 067: Quasigroup completion. http://www.csplib.org/Problems/prob067
40. Régin, J.C.: Generalized Arc Consistency for Global Cardinality Constraint. In: Proceedings of the 13th National Conference on AI (AAAI'96). vol. 1, pp. 209–215 (1996)
41. Rossi, F., van Beek, P., Walsh, T. (eds.): The Handbook of Constraint Programming. Elsevier (2006)
42. Samer, M., Szeider, S.: Tractable cases of the extended global cardinality constraint. Constraints 16, 1–24 (2011)
43. Wallace, M.: Practical applications of constraint programming. Constraints 1, 139–168 (1996)
44. Wallace, M., Novello, S., Schimpf, J.: ECLiPSe: A platform for constraint logic programming. ICL Systems Journal 12, 137–158 (1997)