CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	Replaceability and a simple substitutability hierarchy for constraint satisfaction problems
Authors	Wallace, Richard J.;Freuder, Eugene C.
Publication date	2025-06-14
Original Citation	Wallace, R. J. and Freuder, E. C. (2025) 'Replaceability and a simple substitutability hierarchy for constraint satisfaction problems', Journal of Experimental & Theoretical Artificial Intelligence, pp.1-48. https://doi.org/10.1080/0952813X.2025.2509181
Type of publication	Article (peer-reviewed);journal-article
Link to publisher's version	10.1080/0952813x.2025.2509181
Rights	© 2025, Informa UK Limited, trading as Taylor & Francis Group. This is an Accepted Manuscript of an item published by Taylor & Francis in Journal of Experimental & Theoretical Artificial Intelligence on 14 June 2025, available online: https://doi.org/10.1080/0952813X.2025.250918
Download date	2026-10-30 12:04:45
Item downloaded from	https://hdl.handle.net/10468/17704

UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Replaceability and a Simple Substitutability Hierarchy for Constraint Satisfaction Problems

Richard J. Wallace[a] and Eugene C. Freuder[b]

Insight Centre for Data Analytics and Department of Computer Science
Western Gateway Building, University College Cork, Cork, Ireland
ORCID:[a]0000-0002-3537-7719 [b]0000-0002-1467-5012

ARTICLE HISTORY
Compiled May 11, 2025

ABSTRACT
Problem simplification is of continuing interest in the field of constraint satisfaction. In this paper we examine properties associated with the basic idea of value substitutability and show how certain forms of substitutability can be organized into a strict hierarchy. One of these properties, here called *replaceability*, has been identified by other authors as being of special interest. We confirm these claims by showing that replaceability is the most general property in this hierarchy that allows local to global inferences. We introduce new algorithms for establishing local (neighbourhood) replaceability that are superior to other types of algorithms designed for this task and demonstrate their effectiveness (values removed) for a variety of constraint types. We show that it is sometimes possible to infer replaceability, leading to appreciable reductions in time required to reduce a problem to irreplaceable domain sets. We also describe a new kind of complexity peak that occurs with neighbourhood replaceability algorithms and analyze this effect.

KEYWORDS
constraint satisfaction; substitutability; replaceability; problem reformulation

1. Introduction

Reformulating constraint satisfaction problems is done for a variety of purposes. To date, a major focus has been on simplifying problems by using inference to remove values or combinations of values in order to reduce the effort to find a solution. To this end, removal of inconsistencies, which will not eliminate any solutions, has been most thoroughly studied. But we can also remove values if we know that other values can serve as replacements in at least some solutions. This is the idea of *substitution*, which is the topic of the present paper. A special case occurs when either of two values can be substituted for the other; however, the important idea for purposes of simplification is substitution and removal.

However, substitutability properties have uses other than simplification that are potentially of equal or greater practical significance. Suppose, for example, that one is dealing with an inventory of parts. For each part, it would be useful to know if there are other parts that could replace it if necessary. Moreover, it would be helpful

CONTACT R. J. Wallace Email::richard.wallace@insight-centre.org

if we knew that for every context in which that part could be used, there was another part that could replace it. If information of this kind could be added to our problem representation, then this could be of considerable value for future planning, inventory monitoring, etc. By the same token, it would also be useful to know which parts were irreplaceable. Hence, it might be worthwhile to identify relations among parts in terms of this property, even if this cannot be done in polynomial time. (See (Weigel & Faltings, 1999) for related ideas.)

The present paper uses problem simplification and search as a means of studying topics such as methods for achieving substitution, and the effects of removing substitutable values. This allows us to characterize substitutability properties with considerable precision and to study features of the resulting problems. The results of this investigation should therefore be relevant to issues raised in the previous paragraph as well as the topic of problem simplification per se.

The first paper on this subject defined two basic properties involving either equivalence or subsumption of single values, called *interchangeability* and *substitutability*, respectively (Freuder, 1991). Since then a large collection of related properties has been proposed, and various dominance and incomparability relations have been established; for a summary of this work, see (Karakashian, Woodward, Choueiry, Prestwich, & Freuder, 2010). Some years ago Bordeaux, Cadoli, and Mancini proposed a general framework for these and other properties (Bordeaux, Cadoli, & Mancini, 2008). They identified what they considered a key concept that they called 'removeability' and that we will refer to as *replaceability*, which is a generalized form of substitutability. Subsequently, Freuder attempted to enlarge this framework even further, using the idea of 'dispensability' (Freuder, 2011), which simply means that removing elements of a problem will not remove all solutions.

It will be appreciated that at least some of these properties, especially those based on equivalence, can be viewed as special forms of symmetry (see (Gent, Petrie, & Puget, 2006) for a discussion). However, since our concern in this paper is with substitution, which can often occur asymmetrically, we will not discuss any possible relations to the literature on symmetry in this paper.

In the present work, which carries on the work of both (Bordeaux et al., 2008) and (Freuder, 2011), we focus on the basic property of substitutability and its generalisations, all of which involve relaxing the conditions under which we may remove a given value (or set of values). The reason for considering this special set of properties, which forms a relatively small subgraph in the network of properties discussed by (Karakashian et al., 2010), is that this set has a well-formed set of interrelations so that they can be arranged in a single series, i.e. a total order, or more simply a hierarchy. For similar reasons, in this paper we do not consider extensions of the substitutability property as described in (Bowen & Likitvivatanavong, 2004) and (Cooper, 2020), in which substitutability depends on more complicated conditions including relations among adjacent variables. Nor do we consider other 'merging' strategies such as the broken triangle property (Cooper, El Mouelhi, Terrioux, & Zanuttini, 2014). To distinguish the properties of concern from other forms of substitution and merging, we refer to them as forms of 'simple substitutability' and the hierarchy they form as a simple substitutability hierarchy, although for brevity we often refer to the latter as the substitutability hierarchy.

The importance of a more limited framework of this sort is that it allows us to better characterise the replaceability property highlighted by (Bordeaux et al., 2008). This is because the hierarchy itself has strong properties from which we can derive propositions that clarify important features of replaceability. In particular, we show

that the latter has certain 'maximal' features within this hierarchy with regard to its computability. This confirms the contention of these authors that this is a property of special significance.

In the following sections we first review earlier work that presented generalisations of substitutability. Then we present our basic framework, centered on the concept of replaceability, but attempting to situate it within existing work. Then we introduce local forms of this property that allow us to devise working algorithms that can achieve these forms. This is followed by extensive experimental tests. We then discuss how replaceability can sometimes be inferred to reduce overall effort and demonstrate the efficiency of some of these methods with a further set of experiments. We also describe a new kind of complexity peak that occurs with neighbourhood replaceability algorithms and analyze its properties. We finish with conclusions and suggestions for future work.

Some of this work appeared in (Freuder & Wallace, 2017; Wallace, 2018).

2. Background Concepts

A constraint satisfaction problem (CSP) involves choosing values for a set of variables, X, which satisfy a set of constraints, C. Each constraint specifies permissible combinations of values for a subset of the variables. Each value selected for variable X_i must come from a domain of values D_i associated with that variable.

More formally, a CSP can be defined as a tuple, (X, D, C) where:

X is a set of *variables*, $X_1, \ldots, X_n$,

D is a set of *domains*, D_i, where each D_i is a set of possible *values* for variable X_i

C is a set of *constraints*. Each C_i belonging to C consists of a particular subset of the variables in X, called the *scope* of the constraint, and a relation R_i defined on an ordering of the scope. Specifically, R_i is a subset of the Cartesian product of the values of the domains of the variables that form the ordering. In the text, this ordering of variables is indicated by an ordering of subscripts; e.g. C_{jk} indicates an X_j, X_k ordering for this binary constraint. In this case, therefore, in tuple (a, b) a is a value in the domain of X_j and b in the domain of X_k.

A choice of values for a subset S of the variables is an *instantiation* of S. If the instantiation is consistent with the constraints involving S, then we say that it *satisfies* those constraints. For example, in the case just given, if (a, b) is one of the tuples in the relation associated with C_{jk}, and a is the instantiation of X_j, then a would satisfy constraint C_{jk}. An instantiation of all the variables, X, which satisfies all the constraints is a *solution*.

CSPs can be characterised in a number of ways. A simple scheme defines a small set of parameters. These include problem size, which is the number of variables (i.e. $|X|$), and (average) domain size $\frac{\sum_{i=1}^{n} |D_i|}{n}$. The network of constraints that forms a CSP can be represented as a graph or hypergraph. In such *constraint graphs*, variables are represented as nodes and constraints as edges between nodes (binary constraints) or hyperedges that include three or more nodes (non-binary constraints). Examples of constraint graphs, where the arity of the constraints is always ≤ 2, are found in Figures 1, 3, 4, and 10. In the case of binary networks such as these, we can define a graph *density*, which is the proportion of arcs in the graph in relation to the total number of possible arcs. Another important parameter is the (average) *tightness* of a constraint; this is the number of value tuples not in R_i as a proportion of the tuples in the Cartesian product. Higher tightness, therefore, is associated with fewer tuples

that satisfy the constraint.

A common method for finding solutions to CSPs involves the backtracking algorithm interleaved with local inference methods. Local inference methods are also often used in a preprocessing step before search begins. Such methods include procedures for establishing various forms of local consistency as well as forms of substitutability, which is the topic of the present paper. Finally, we note that search as well as inference methods can often be improved by heuristics, i.e. rules of thumb, that guide choices among elements of the problem that need to be processed. Among these are variable ordering heuristics that select which variable to instantiate at each stage of backtrack search.

3. Generalisations of Substitutability

The basic property of substitutability in CSPs was characterised in (Freuder, 1991). Here, we will use an equivalent definition that corresponds to later definitions in which the focus is on the value discarded rather than the value that can be substituted.

Definition 1. Given a value v in domain D_i of variable X_i, if for every solution in which v appears, we can substitute the same value u and still have a solution, then v has the property of *substitutability* and value u can be substituted for v.

The same work introduced a local form of substitutability, called *neighbourhood substitutability*.

Definition 2. (based on (Freuder, 1991)) For two values v and u belonging to the domain of variable X_i, v has the property of *neighbourhood substitutability* with respect to u iff for every constraint on X_i, and for each tuple in that constraint containing v, if u is substituted for v, the resulting tuple also satisfies that constraint.

In this paper we also sometimes say that v is *substitutable* if it has the substitutability properties specified in Definitions 1 and 2.

This latter property had two important features: (i) neighbourhood substitutability is sufficient (but not necessary) for full substitutability, (ii) neighbourhood substitutability can be determined in polynomial time. Proofs of these properties can be found in (Freuder, 1991).

Given this interesting property, one important question is whether we can isolate its general features. In addition to elucidating the original property itself, this may allow us to identify more general properties that may allow even greater degrees of problem simplification.

In the past, there have been a few efforts in this direction. The earliest is the work of Jeavons et al., who defined what they called 'generalized substitutability' (Jeavons, Cohen, & Cooper, 1994). Later Bordeaux et al. defined a similar but simpler concept that they called 'removability' (Bordeaux et al., 2008). In both cases, the basic idea is that, given a value v that can appear in some solution, then for any solution that v appears in there is some other value that can be substituted for v, to give another valid solution. (A more formal definition is given in the next section and later the relation between the two approaches to this idea is discussed at length.)

In (Freuder, 2011), the more general concept of dispensability was introduced.

Definition 3. A value is *dispensable* if removing it will not remove all solutions to the problem. A set of values is dispensable if removing them all will still not remove all solutions to the problem.

However, since this idea goes beyond the requirement of substition in any form (an

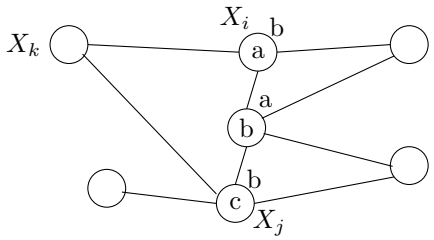


Figure 1. Example showing 3-tuple substitutability without substitutability with respect to single values in the tuple. In this figure, circles represent variables, lines constraints between variables. (So, for example, constraint C_{ik} holds between variables X_i and X_k.) Values to be substituted are inside the circles, substituted values above and to the right of each circle.

instantiation can be dispensable without there being any values that can be substituted for it), it is tangential to the main concerns of the present paper. Of more immediate relevance is the terminological problem raised by the introduction of this new concept. In his paper Freuder pointed out that, since the ordinary meaning of removability is synonymous with dispensability, the former term is somewhat misleading in its original context. He suggested the term 'onto-substitutability' instead. Here, we will use what we think is a more perspicuous term: 'replaceability', which seems to have exactly the right connotations.

More recently (Likitvivatanavong & Yap, 2013) also discussed the concept of replaceability. They introduced a generalisation that they called a substitutable set. To this end, they first defined substitutability by a set:

Definition 4. (Based on (Likitvivatanavong & Yap, 2013)) A value $v \in D_i$ is substitutable by a set of values $S \subseteq D_i \backslash v$ if for every solution that includes v, there is a value $w \in S$ such that a solution is obtained when w is substituted for v.

And they then extended this idea in a straightforward way:

Definition 5. (From (Likitvivatanavong & Yap, 2013)) A set of values $S \subseteq D_i$ is substitutable by set $T \subseteq D_i$ iff for any value $v \in S$, v is substitutable by T (in the sense of Definition 4).

However, this is still not as general as the Jeavons et al. concept (described below) since it deals with values in only one domain.

In this paper we will not make much use of generalisations of substitutability that involve k-tuples of values with $k > 1$ and which can involve sets of values or tuples to be replaced. This is because they entail complexities in computation and representation that will make it difficult to use them in practice. Here, we illustrate some issues when k-tuples are being considered for replacement.

To begin with, we can show that substitutability in terms of k-tuples over a variable set R does not imply that individual values in the set are substitutable. This can be done by example (see Figure 1).

Here, all domains have values a, b, c. Suppose that the 3-tuple (b, a, b) can be substituted for (a, b, c), where the assignments are made as shown in the figure. Also, suppose that constraint C_{ik} consists of these tuples: $\{(a, b), (a, c), (b, b)\}$, while constraint C_{jk} consists of these tuples: $\{(b, b), (b, c), (c, a), (c, b)\}$. An examination of the tuples will show that the only value in D_k that supports both a in D_i and c in D_j is b. In addition, the listed tuples are consistent with the substitutability of (b, a, b). However, if we consider $a \in D_i$ by itself, then we can ascertain that it is *not* substitutable with respect to b. Since these arguments can be extended to the other variables associated with this three-tuple, we have a situation where a k-tuple can be replaced while this is not true

for any individual value taken alone. In other words, the property of substitutability when applied to a k-tuple is not decomposable.

One result of this is that ascertaining that a set of k-tuples can be substituted for another set is likely to involve considerable effort with only marginal results. Consequently, it is reasonable to focus on substitutability properties with respect to single values, which is what we will do for most of the remainder of the paper.

4. A Simple Substitutability Hierarchy

We now define the other substitutability properties in our hierarchy. First, we define the central concept:

Definition 6. (After (Bordeaux et al., 2008)) A value $v \in D_i$ is *replaceable* if for every solution that contains v, there is a different value $w \in D_i$ that can be substituted for v to make a valid solution.

(Note that this is equivalent to substitutability à la Likitvivatanavong and Yap, cf. Definition 4.) In other words, a value is replaceable if a value can be found to substitute for it in every solution that it participates in, but it is substitutable only if there is a single value that can be substituted for it in every solution. Obviously, replaceability is weaker than the original substitutability property in that the latter implies replaceability, while replaceability does not imply substitutability. At the same time, it is more general since any values that can be removed on the basis of substitutability are also replaceable, but not vice versa.

Still weaker (and more general) forms of substitutability can be defined.

Definition 7. Let $Sol_v(C)$ be the set of solutions where value v in domain D_i is assigned to variable X_i and let s be the number of such solutions. Then v is *k-restricted replaceable*, where k is some positive number ≥ 1, if either of the following holds: (i) when $s \geq k$, then for each solution in some set of k solutions in which v appears, we can substitute some other value $w \in D_i$ and still have a solution. (ii) when $s < k$, then for all solutions where v appears we can substitute some other value $w \in D_i$ and still have a solution.

Note that, as with replaceability, the substituted value w need not be the same for all solutions in the designated set of size k. For brevity, k-restricted replaceability is sometimes referred to as restricted replaceability.

Definition 8. A value v in domain D_i is *minimally replaceable* if either of the following holds: (i) there is at least one solution where v is assigned to variable X_i and for this solution we can substitute a different value $w \in D_i$ and still have a solution. (ii) there are no solutions where v is assigned to variable X_i.

Minimal replaceability is therefore a special limiting case of restricted replaceability, where $k = 1$.

The hierarchy formed by these properties is depicted in Figure 2. The properties are also listed in Table 1 along with definitions in mathematical notation. In this table we adopt the functional notation used by (Bordeaux et al., 2008), although instead of using S, the set of all possible tuples of n values (where n is the number of variables in the problem), we use $Sol(C)$, the set of tuples of S that satisfy all the constraints of the problem, aka the solution set. x and a are the variable and value, respectively, with the given property. Then, $Sol(C)[x = a]$ is the set of solutions where a is assigned to x. In addition, $D(x)$ is the domain of x, t and t' are n-valued tuples, b is a value

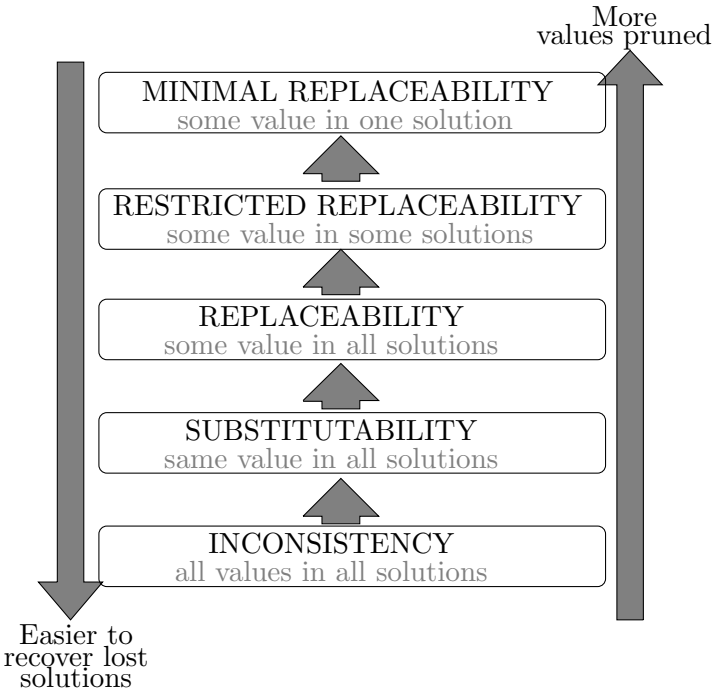


Figure 2. The simple substitutability hierarchy used in this paper.

in $D(x)$ different from a, and T is a set of tuples in S. Finally, t_x and t_{-x} are the projections of tuple t on x and on all variables except x, respectively.

Inconsistency is also included in the hierarchy. It is viewed as an extreme case of substitutability, since if value v is inconsistent then any value, consistent or inconsistent, can be substituted for it without losing any solution. (In fact, (Jeavons et al., 1994) have shown that consistency enforcement is always subsumed by a substitution operation, so it fits naturally into the hierarchy; cf. their Proposition 9.) Since inconsistent values do not appear in any solution, inconsistency is subsumed by the more general properties in the hierarchy as well.

Definition 9. A value $v \in D_i$ is *inconsistent* if v does not appear in any solution.

Thus, the weakening and generality mentioned above hold over the entire hierarchy. That is, each property in the hierarchy is implied by all those below it, and it in turn implies those above it in the hierarchy. As we proceed up the hierarchy, we can also remove more instantiations. But it also becomes harder to recover the solutions lost by discarding values.

Proposition 1. If a value can be discarded because a given property in the substitutability hierarchy applies to it, then it can be discarded on the basis of any property above that property in the hierarchy.

There is an important division in the series. Up to and including replaceability, the property in question is defined with respect to *all* solutions in which value v appears, i.e. all solutions supported by a replaceable value v are still supported.

Proposition 2. Within this substitutability hierarchy, replaceability is the most general form of substitutability for which all solutions that include a given value a from the domain of variable X_i can be transformed into another solution by replacing a with another value from the same domain.

Table 1. Simple Substitutability Properties

name	definition
inconsistency$(Sol(C), x, a)$	$\forall t \in Sol(C)\ t_x \neq a$
substitutability$(Sol(C), x, a)$	$\exists b \in D(x)\ (b \neq a\ \wedge\ \forall t \in Sol(C)\ (t_x = a) \rightarrow \exists t' \in Sol(C)\ (t'_x = b\ \wedge\ t'_{-x} = t_{-x}))$
replaceability$(Sol(C), x, a)$	$\forall t \in Sol(C)\ (t_x = a\ \rightarrow\ (\exists b \in D(x)\ (b \neq a\ \wedge \exists t' \in Sol(C)\ (t_x = b\ \wedge\ t'_{-x} = t_{-x})))$
k-restricted-replaceability$(Sol(C), x, a, k)$	$\exists T \subseteq Sol(C)[x = a]\ (\|T\| = min(k, \|Sol(C)[x = a]\|) \wedge \forall t \in T\ (\exists t' \in Sol(C)\ (t'_x \neq a\ \wedge\ t'_{-x} = t_{-x}))$
minimal-replaceability$(Sol(C), x, a)$	$\exists T \subseteq Sol(C)[x = a]\ (\|T\| = min(1, \|Sol(C)[x = a]\|) \wedge \forall t \in T\ (\exists t' \in Sol(C)\ (t'_x \neq a\ \wedge\ t'_{-x} = t_{-x}))$

Proof. By definition, if value a is replaceable then all its associated solutions remain solutions when a is replaced by at least one other value in the same domain. If even one such solution cannot be repaired in this fashion after discarding value a, then the associated property is restricted replaceability, which does not guarantee that all solutions can be transformed into valid solutions in this manner. □

It can therefore be said that replaceability is a 'maximal property' within the substitutability hierarchy with respect to this feature.

Now we will show that the 'generalized substitutability' property of Jeavons et al. is essentially the concept of replaceability in a somewhat different form. These authors give their definition of generalized substitutability in terms of two functions σ and π, which are selection and projection functions, respectively. Given two sets of variables X and R, where $R \subseteq X$, if we consider a set of instantiations S of X and a set of instantiations T of R, then $\sigma_T(S)$ selects the set of tuples in S whose projection on R matches some instantiation in T, and π_R applied to the output of σ restricts the instantiations to those of the variables in R. Their definition is as follows:

Definition 10. Given a problem P with variables X, and two sets of instantiations T_1 and T_2 of $R \subseteq X$, T_2 can be substituted for T_1 (in their generalized sense) if

$$\pi_{X-R}(\sigma_{T_1}(\mathrm{Sol}(P)))\ \subseteq\ \pi_{X-R}(\sigma_{T_2}(\mathrm{Sol}(P)))$$

In words, T_2 can be substituted for T_1 if the instantiations for variables other than those in R (namely, $X - R$) that are associated with the selection based on T_2 form a superset of the instantiations for $X - R$ associated with the selection based on T_1.

That this form of substitutability is a generalized form of replaceability in our terms can be seen by setting T_1 in the definition to a single instantiation of R. (Therefore, if $|R| = 1$ T_1 is a single value.) In this case, T_2 becomes a set of instantiations of R that support all of the instantiations elsewhere in the problem that T_1 supports. This is the same as saying that one can replace T_1 in any solution with some other instantiation. Thus we have the following equivalence.

Proposition 3. Given Definition 10, if T_1 is a singleton, then the instantiation in T_1 is substitutable by T_2 in the sense of (Jeavons et al., 1994) iff it is replaceable.

Proof. If the instantiation in T_1 is replaceable, then all of the projections on the remaining variables in the set of solutions in which this instantiation occurs can be supported by some set of instantiations of R. If this latter set is chosen as T_2 then Definition 10 is satisfied. Conversely, any T_2 that can be substituted in the sense of (Jeavons et al., 1994) for the instantiation in T_1 must contain a set of instantiations that can support any projection on $X - R$ consistent with the instantiation in T_1 by definition. But this is equivalent to the statement that these instantiations can replace the instantiation in T_1 in any solution in which the latter appears. □

The concept in (Jeavons et al., 1994) generalizes the present concept of replaceability in two ways. In the first place, T_1 is no longer necessarily a single instantiation, but a set, as already noted. Secondly, the formulation of (Jeavons et al., 1994) is more general in some ways with respect to the possible T_2 sets associated with a given T_1. For example, taking single values, consider the case where b can be substituted for a in all solutions, i.e. $T_1 = a$ and $T_2 = b$. Suppose further that another value c in the same domain has no solutions in common with a. Then (Jeavons et al., 1994) would allow the set $\{b, c\}$ to substitute for a. While this situation is not ruled out by our formulation, it does involve a value that is 'irrelevant' to the definition of replaceability given above (and by (Bordeaux et al., 2008)). This leads to the following observation.

There is a difference in perspective between (Jeavons et al., 1994) and the present work that is potentially significant. They view the replaceability property from the point of view of the instantiations that can substitute for a given set of instantiations. We and (Bordeaux et al., 2008) on the other hand, view replaceability in terms of the instantiations that can be discarded on this basis. Despite the elegance and greater generality of the former perspective, in some ways the latter perspective is more perspicuous. It may also lead to more focused (and therefore effective) algorithms, as will be seen later in this paper.

Removing instantiations based on these properties, either in preprocessing or dynamically during search, can greatly simplify a problem. In general, however, determining various forms of substitutability can be at least as difficult as finding a solution. One way to address this is to consider restricted classes of problems, where the computation of forms of substitutability is tractable (Bordeaux et al., 2008). We focus here instead on situations in which a substitutability property with respect to a subproblem implies this property for the full problem, and thus the complexity is limited by the size of the subproblems we consider.

5. Substitutability Properties and Local Reasoning

Studies of inconsistency and substitutability have shown that tractable forms of these properties exist based on reasoning about subproblems that involve only a subset of the variables in the original problem. Here, we show that some of these ideas can be extended to more general forms of substitutability. For brevity, we restrict the discussion to subproblems based on closure (defined below), although it is also possible to reason about induced subproblems in a way that is analogous to k-consistency and k-interchangeability (Freuder, 2011). However, it is not yet clear that the latter ideas can yield effective algorithms.

Bordeaux et al. claimed that replaceability cannot be 'detected efficiently, but incompletely, through local reasoning'; this, they said, justifies the extensive use of properties like inconsistency and substitutability, which can (Bordeaux et al., 2008). That paper also raised 'an interesting open issue: do there exist new (i.e., other than substitutability and inconsistency) properties for which local reasoning is sound and which imply removability'. However, they used a narrow definition of 'local reasoning'. Thus, their definition of soundness requires one to consider 'all subsets of constraints $C_1 \subseteq C, ..., C_k \subseteq C$ such that $\bigcup_{i \in 1...k} C_i = C$'. Soundness itself requires that for all such collections of subsets, if the property holds for all (or for some properties for at least one of them), then it also holds for the complete problem. This definition leaves open the possibility that there may be particular collections of subsets for which local reasoning is sound. Here we show that for 'closure subproblems' we can define sound

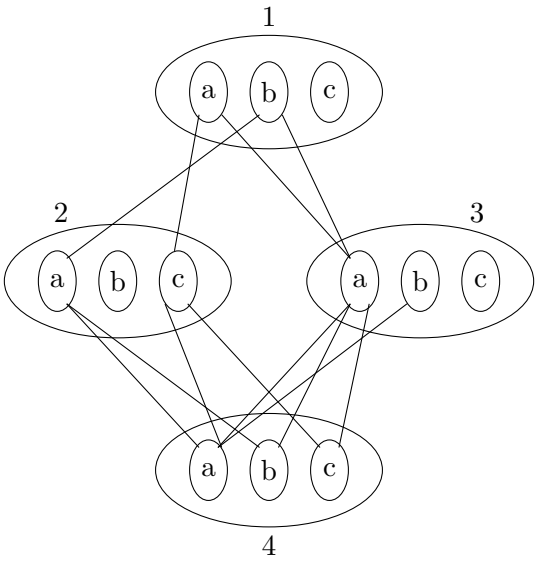


Figure 3. Example to illustrate the relation between neighbourhood and full replaceability. Variables numbered 1 to 4; domain values shown within the small elipses. Lines between small elipses indicate consistency or support. Further explanation in text.

methods that can be applied to properties as general as replaceability.

Definition 11. Let S be a subset of the variables of problem P, and P_S be the subproblem induced by those variables. The *closure* of P_S, $C(P_S)$, is the subproblem induced by S and all the variables that share a constraint with a variable in S. Call the variables of S the *core variables*, and the others in $C(P_S)$ the *frontier variables.*

Note that according to this definition, $C(P_S)$ includes constraints whose scope is restricted to the frontier as well as those that include both core and frontier variables.

In the following definition and proposition, we will generalize the notion of a replaceable value to that of a replaceable instantiation of a set of variables before narrowing the focus again to assignments to single variables.

Definition 12. An instantiation v of a set of variables S is replaceable with respect to a set of variables T, $S \subseteq T$, if for every valid instantiation V of T that includes v, there is some other instantiation u of S that differs from v in at least one value and which, together with the values of $T - S$ in V, forms a valid instantiation of T.

Definition 13. An instantiation v of variables S is *closure-replaceable* if it is replaceable with respect to the closure of the subproblem induced by S. That is, every consistent assignment to the closure of S that includes the k-tuple v can be replaced by some other instantiation u of S to form another consistent assignment to the closure of S.

Closure-replaceability is related to the concept of local substitutability defined in (Jeavons et al., 1994), which again is a replaceability concept. Their concept refers to a constraint c and all other constraints c' such that the intersection of the scope of c and the scope of c' is non-null. Since our variable-based concept of closure includes any set of (core) variables together with all variables that belong to constraints whose scopes include variables within the core, the present concept is in some respects more general. On the other hand, since the (Jeavons et al., 1994) concept considers sets of instantiations that might be replaced, in this respect it is the more general concept.

Proposition 4. Closure-replaceability implies replaceability.

Proof. The basic idea is that it is necessary and sufficient to have 'support' in the core for every instantiation of the frontier that can appear in a solution of the closure.

Any solution of the entire problem must contain a solution of the closure, and the rest of the problem only interacts directly with the closure through the frontier. Consider a problem P, and an instantiation v for variables S that is closure-replaceable, where $C(P_S)$ is the closure. Suppose sp is a solution of P that includes v; sp must also contain a solution, $scps$, to $C(P_S)$ that includes v. Since v is replaceable with respect to $C(P_S)$, there is some other instantiation u of S that can be substituted for v in $scps$ yielding another solution for $C(P_S)$. When we substitute u for v in sp we obtain another solution for P, since the values in the frontier of $C(P_S)$ have not changed. □

Although the closure can be the entire problem, often that will not be the case. Consider, for example, binary CSP's where S consists of a single variable. The closure will be the subproblem induced by that variable and all the neighbouring variables in the constraint graph. If the degree of the constraint graph is bounded by some constant k, or if we restrict our attention to variables with k or fewer neighbours, then the complexity of the effort required to look for closure-replaceable values is correspondingly bounded.

Definition 14. An instantiation of a single variable X is *neighbourhood replaceable* if it is replaceable with respect to the closure of X.

The relationship between closure replaceability, specifically neighbourhood replaceability, and full replaceability is illustrated in Figure 3. Here value a in the domain of variable 4 is replaceable. It participates in two solutions, {b, a, a, a} and {a, c, a, a} (for variables 1, 2, 3, 4, respectively). In the first, 4/a can be replaced by 4/b, and in the second, 4/a can be replaced by 4/c. At the same time 4/a is not neighbourhood replaceable because the neighbourhood instantiations {a, b, a} and {c, b, a} (for variables 2, 3, 4, respectively) cannot be replaced by any other value in the domain of variable 4. This shows that neighbourhood replaceability is not necessary for full replaceability. Now, suppose that the support between 3/b and 4/a did not hold. In this case, 4/a would be neighbourhood replaceable, and in accordance with Proposition 4, it is now also be replaceable with respect to full solutions.

The next proposition suggests that relations analogous to that of Proposition 4 will not be possible for properties in this substitutability hierarchy that are more general than replaceability.

Proposition 5. If a (simple) substitutability property of instantiation I of S does not require that for every solution to the closure $C(P_S)$ that contains I there is an instantiation J of S such that the assignment to the closure obtained by swapping J for I is also a solution to $C(P_S)$, then if the property holds for I in $C(P_S)$, it does not necessarily hold for I with respect to the entire problem P that contains S. In other words, if the property holds for the closure with respect to I, this does not imply that it holds for the entire problem.

Proof. If an instantiation of subset S is part of a solution to closure $C(P_S)$ for which no substitution can be made, then if this is the only closure solution that can be extended to a full solution, the property will not hold for the full problem. This situation can obviously be obtained by construction. □

Corollary 1. Replaceability is the most general property in this substitutability hierarchy that can be inferred from a closure to the entire problem. In other words, it is the maximal property in this hierarchy for which this feature always holds.

It is important to consider relations between neighbourhood replaceability and certain kinds of neighbourhood consistencies that have been proposed in the literature, since the latter also involve closures. One of these is a type of singleton arc consistency (SAC) (Debruyne & Bessière, 1997) called neighbourhood singleton arc consistency. In

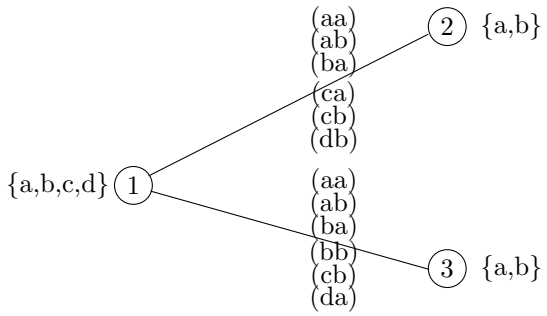


Figure 4. Example showing that more than one fixpoint is possible when all replaceable values are removed (domains shown in brackets, constraint tuples in parentheses).

this form of SAC, arc consistency is established with respect to the subgraph formed by a variable and its neighbours. Here again S is a single variable, X_i, and each of its values is considered singly (Wallace, 2015). If, for a given value a in the domain of X_i, arc consistency processing removes all values from a domain of a variable in the neighbourhood, then this value is neighbourhood inconsistent and can be removed. The other is a form of neighbourhood consistency stronger than AC called neighbourhood inverse consistency (NIC) (Freuder & Elfe, 1996). For a given value a in the domain of X_i to be NIC, there must be at least one set of assignments to each of the neighbouring variables that is consistent with a as well as internally consistent.

In the algorithms described below, it can be readily seen that since checking for neighbourhood replaceability requires an all-solutions search in the neighbourhood subgraph, then if no solution is found at all the value cannot be NIC. In addition, if a value is neighbourhood irreplaceable, it must also be neighbourhood singleton arc consistent, since the latter is a necessary condition for there being a neighbourhood solution. This gives the following.

Proposition 6. If in problem P all neighbourhood replaceable values are discarded, then that problem is neighbourhood singleton arc consistent as well as neighbourhood inverse consistent.

Although replaceability algorithms terminate with a set of irreplaceable values, these sets are not unique. In other words, replaceability algorithms do not have unique fixpoints. This can be shown by example, as in Figure 4. In this problem, if we begin with variable 1 we can either replace value a with b, c, d or vice versa. This will then yield different fixpoints. In fact, the same situation holds for substitutability; see (Cooper, 1997). Cooper also showed that the CSPs produced by different sequences of substitutions are isomorphic; the same example in Figure 4 shows that this is not necessarily the case for replaceability.

6. Neighbourhood Replaceability Algorithms

As shown in the previous section, effective local to global reasoning is possible even with a property as general as replaceability. Hence, the strategy we will focus on in this section involves using closure subproblems to deduce replaceability.

There seem to be two general approaches that one can take for computing replaceability and removing all neighbourhood replaceable values. In the first, which is introduced here, replaceability can be computed locally in a manner analogous to the local computation of neighbourhood inverse consistency (Freuder & Elfe, 1996).

```
1    Repeat
2        Set no-change to true
3        Set Q to list of all variables
4        While not empty Q
5            Remove variable Xi from Q; set S to neighbours of Xi
6            For each value v in domain of Xi
7                Set domain of Xi to {v}
8                If arc-inconsistent(Xi ∪ S) or replaceable(v,V\{v},S)
9                    Remove v from domain
10                   Set no-change to false
11           If domain of Xi is empty set failure to true
12   Until failure or no-change
```

Figure 5. CNR-1 algorithm for neighbourhood replaceability. V is full domain of variable X_i.

Essentially, this involves an all-solutions search for each value within its variable's neighbourhood. The second approach depends heavily on inferences about the number of neighbourhood tuples associated with a given value and the number available to replace it. This is exemplified by the algorithm described by Likitvivatanavong and Yap (Likitvivatanavong & Yap, 2013). In this section we will describe two algorithms based on the first approach and a slightly altered version of the Likitvivatanavong-Yap algorithm. (An algorithm has been proposed for finding the generalized form specified by (Jeavons et al., 1994), but since it is not oriented specifically toward finding single replaceable values, and since it appears to be fairly inefficient (see below), it is not described here.)

The algorithms for establishing neighbourhood replaceability described in this paper will be designated as 'consistent neighbourhood replaceability' (CNR) algorithms because they remove all values that are locally replaceable including arc-inconsistent values. As shown in the previous section, neighbourhood replaceability subsumes (and therefore dominates with respect to value removal) NIC and neighbourhood SAC.

6.1. Algorithms based on all-solutions search

The first two algorithms we will describe use depth-first search within a neighbourhood to determine replaceability. Since this is necessarily an all-solutions search, the question raised is whether this method can be done efficiently, or whether there are better approaches that avoid repeated search. But first we describe two CNR algorithms based on this procedure.

The first algorithm, called CNR-1, uses an AC-1 style procedure in which all values in the problem are repeatedly tested for replaceability until no value is discarded (Mackworth, 1977). In the implementation of this algorithm, the replaceable procedure uses a MAC-style search (MAC=maintained arc consistency) to find all solutions for the subproblem (Sabin & Freuder, 1994), and for each solution it seeks a value from the current domain that can replace value v. Pseudocode for this algorithm is shown in Figure 5; pseudocode for the subroutine replaceable is shown in Figure 7.

For binary CSPs, we calculate the complexity of CNR-1 as follows. To simplify the analysis, we assume a constant domain size, although d_{max} could be substituted if domain sizes are variable. Each iteration of the repeat loop requires nd steps, where n is the number of variables and d is the domain size. In the worst case, only one value is deleted in each iteration, so there can be as many as nd iterations. For the complexity of search, we will use the complexity of backtrack search, $O(d^n e)$ (Tsang,

```
1    Set Q to list of all variables
2    While not empty Q and not failure
3        Remove variable X_i from Q; set S to neighbours of X_i
4        For each value v in domain of X_i
5            Set domain of X_i to {v}
6            If arc-inconsistent(X_i ∪ S) or replaceable(v,V\{v},S)
7                Remove v from domain
8                Put neighbours of X_i in Q if not there already
9        If domain of X_i is empty set failure to true
```

Figure 6. CNRQ algorithm for neighbourhood replaceability.V is full domain of variable X_i.

1993), using n_{ne} (maximum size of a neighbourhood) instead of n and e_{ne} (maximum number of neighbourhood constraints) instead of e. This gives $n^2d^2d^{n_{ne}}e_{ne}$, which is $O(n^2d^{n_{ne}+2}e_{ne})$. For general CSPs, the e_{ne} term will also contain k-ary constraints, so the same expression can be used to express the complexity of the algorithm.

Proposition 7. The CNR-1 algorithm is sound in that it is correct, complete, and always terminates.

Proof. *Correctness.* Since neighbourhood replaceability is tested directly, the algorithm will only discard values that have this property.

Completeness. Since CNR-1 checks all values in the problem, any values that are replaceable in the original problem will be discovered. In addition, since CNR-1 checks all values in the problem subsequent to any deletion (i.e. in the next repeat), any values that become replaceable because of deletions will be discovered in the next iteration of the repeat loop.

Termination. Since by definition the problem has only a finite number of values that might be replaced, the algorithm will always terminate. □

CNR-1 can be compared to Algorithm 12 of (Jeavons et al., 1994). The latter algorithm also consists of an outer repeat loop, which executes until no changes are made to the problem. The inner (for) loops consider each constraint c and each tuple t in that constraint, respectively, and tests whether $c - t$ can be substituted for c in the subproblem $P|_{\bar{c}}$ formed by all the variables in all the constraints that are adjacent to c. Clearly, this requires finding all viable extensions of constraint c in $P|_{\bar{c}}$ to see if they are supported by $c - t$. It seems fair to say that this is a less focused approach than one which at each step considers a single value and its neighbourhood extensions to see if that value can be discarded.

A second algorithm, called CNRQ, uses a queue updating mechanism in place of the repeat loop used by CNR-1. Pseudocode for this algorithm is shown in Figure 6.

Proposition 8. The CNRQ procedure is sound and achieves the same results as CNR-1.

Proof. *Correctness.* Since neighbourhood replaceability is tested directly, the algorithm will only discard values that have this property.

Completeness. If values are replaceable on a given pass, since this property is tested for each value (line 6), such values will always be discovered. The only way in which a formerly irreplaceable value could become replaceable is for neighbourhood solutions in which it participates to be lost, so that for the remaining solutions it is replaceable. This can only happen if values in neighbouring variables are discarded. But if this happens the variable with this value in its domain is put back on the queue; hence the alteration in status of the value will be detected.

```
Replaceable(v,V_o,S)   % V_o is domain of X without v
1    Set irreplaceable to false
2    Set done to false
3    While not done ⋀ not irreplaceable
4        nextsol = DFS(v,S)
5        If not nextsol
6            Set done to true
7        Else
8            Set V to V_o
9            Set replaceable to false
10           While not empty(V) ⋀ not replaceable
11               Remove v' from V; Set V to V - v'
12               Set fail to false
13               For each constraint C in subgraph P(S)
14                   fail = check-constraint(v',C,nextsol)
15                   If fail break
16               If not fail
17                   Set replaceable to true
18           If not replaceable
19               Set irreplaceable to true
20   Return (not irreplaceable)
```

Figure 7. Algorithm to determine neighbourhood replaceability.

Termination. Since all neighbourhood replaceable values will be detected the algorithm will terminate when this is accomplished and the queue is emptied or when a domain is wiped out. □

Since each time a variable is taken from the queue, all d values are tested, we can think of the original queue as having nd values. Supposing that only one value is deleted each time an entire domain is checked for consistency, then each addition to the queue includes no more than e_{ne} variables times d values. In this case, we can compute a more precise value for the number of queue additions using e_{ne_i} for the neighbouring constraints of variable X_i. This gives a total number of queue additions of

$$\sum_{i=1}^{n} d(de_{ne_i}) \;=\; d^2 \sum_{i=1}^{n} e_{ne_i} \;=\; 2d^2e$$

(cf. (Mackworth & Freuder, 1985)). So the total number of operations is bounded by $(nd + 2d^2e)d^{n_{ne}}e_{ne}$ giving a complexity of $O(ed^{n_{ne}+2}e_{ne})$.

6.2. An algorithm based on solution counting

The third algorithm is from (Likitvivatanavong & Yap, 2013). The idea behind this algorithm is to count the number of neighbourhood tuples with values other than the value being considered for replacement and match that against the number of neighbourhood tuples containing the value being checked. Under specified conditions, if the former count is greater than the latter, this implies that the value being checked is replaceable.

Since the procedure described by the authors processes only a single value, we adopted the CNRQ top-level queue for handling the entire problem. In addition we substituted NIC for the maxRPC algorithm in the original description; this ensured

```
1   Set up merge tables for entire problem
2   Set Q to list of all variables
3   While not empty Q
4       Remove variable Xi from Q; set S to neighbours of Xi
5           For each value v in domain of Xi
6               Establish neighbourhood inverse consistency
7               If any merge list is singleton v        % value is irreplaceable
8                   continue
9               goal-count ← |{assignments of {Xi} ∪ S containing v}|
10              ub-count ← |{assignments of {Xi} ∪ S not containing v}|
11              If ub-count < goal-count                % value is irreplaceable
12                  continue
13              Choose any constraint on Xi
14              While potential tuples remain
15                  Form next tuple that can replace v from merge-lists
16                      and save if not already in store
17                  If |tuples in store| = goal-count
18                      remove value and continue
19          If values removed set Q to union of Q and S
```

Figure 8. Likitvivatanavong-Yap (LY) algorithm for neighbourhood replaceability embedded within a top-level AC-3 style queue. '%' denotes comments.

compatibility with the other algorithms. Pseudocode for the resulting algorithm is shown in Figure 8.

The algorithm begins with some initialisation steps. The most elaborate is the setting up of *merge tables* for each variable in the problem. For expository purposes, we will consider only problems with binary constraints. In this case, the tables for each variable X show the values in its domain that are supported by each value in the domain of each adjacent variable. Suppose, for example, that there is only one variable W adjacent to X, each variable has three values, 1, 2 and 3, and the constraint is an inequality constraint. Then the merge table would have the form,

$$\begin{array}{l} \{2\ 3\}\,1 \\ \{1\ 3\}\,2 \\ \{1\ 2\}\,3 \end{array}$$

where the values in brackets are those in the domain of X that go with the value in the domain of W, which is indicated on the right. In their paper, the values in brackets are called *merge sets*. (In practice, the values of W can be represented implicitly by an array index.)

In the main procedure, at each step a variable is taken off the queue and all values in its domain are checked for replaceability. After removing the neighbourhood-inconsistent values (line 6), this algorithm carries out a series of steps, each of which determines if the value being tested is irreplaceable or replaceable. In the first test the merge sets are checked to determine if the present value is contained in a singleton merge set; in this case, the value is irreplaceable. Prior to the next two tests, two values are computed: the number of neighbourhood tuples associated with the value being tested (goal-count) and the number of neighbourhood tuples associated with alternative values (ub-count). (In the merge table above, for value 1 the goal-count and ub-count are both 2.) Then, the next test checks whether the ub-count is less than the goal-count; if it is, then the value is irreplaceable. Finally, viable tuples are

collected for the values that could substitute for the value being tested; if ever their number equals the goal-count, then the value being tested is replaceable.

The rationale for the latter test is as follows. Each new tuple in the store is one that supports v, the value being considered for replacement, as well as some other value in the domain of X. If the number of such tuples equals the goal-count, i.e. the total number of tuples that support v, then all supporting tuples have been checked for replaceability.

The final test is the most elaborate since it must calculate a union of tuples. In the present work two methods were tried. The first uses an array with dimensions Number of Adjacent Variables $\times$ Domain Size $\times$ Number of Tuples. Array entries are TRUE or FALSE (t or nil) depending on whether or not the kth value in tuple t has value a. This allows fast checking, but can require $O(|\{\text{neighbourhood tuples}\}|)$ tuple checks. The second method uses a trie structure (Fredkin, 1960) in which if the kth value in tuple t has value a, then at that level the (domain size) array will point to another array representing the next value in the tuple (or if a is the last value, the array entry will have the value TRUE). This allows for fast checking, although there is a proliferation of small arrays each time a value is checked for replaceability. Since the second method was found to improve performance by at least an order of magnitude, this was the method used in experimental tests.

Like the other CNR algorithms, in the present version of CNR-LY the queue of variable-value pairs has $O(ed^3)$ entries. For each value the complexity is dominated by NIC, which is $O(d^{n_n} e_n)$. Therefore, the asymptotic complexity is similar to the other CNR algorithms. However, as we will see, because of the calculations involved in the counting process and the updating of merge sets required after value removal, this algorithm is relatively inefficient in practice.

7. Experimental Analyses

As indicated in the Introduction, we expect that from a practical perspective, higher order properties such as replaceability will be most useful for enhancing problem representations as well as for problem compilation and other specialized tasks. Therefore, the major contribution of the present work is theoretical and has been presented in the previous sections.

However, we can perform experiments using simple one-solution search to answer some important questions concerning replaceability algorithms, questions that at this time can't be clearly answered with formal analysis:

- Are algorithms that employ depth-first all-solutions search competitive against existing algorithms that use counting strategies?
- To what degree can replaceability be inferred and does this have a significant effect on efficiency?
- How effective are replaceability algorithms for constraints of different types?

We can also address questions concerning the tradeoff between efficiency and effectiveness. Here, efficiency simply means the time required to establish a certain property, while effectiveness refers to the degree of alteration of the original problem. In the present case, this is the number of values removed from the problem. The present experiments allow us to compare different algorithms with respect to these two features.

7.1. Methods

To answer the questions just posed, three types of problems were used:

(1) random unstructured problems,
(2) random structured problems,
(3) benchmark problems and problems derived from benchmarks.

These included both binary CSPs and CSPs with k-ary constraints (including global constraints). For many categories, problems could be generated in-house, so that a moderately large sample of problems could be tested rather than a few benchmarks. This also allowed us to vary problem parameters in order to evaluate their effects on the basic tradeoff between effectiveness and efficiency. The range of problems used also allowed us to examine the amenability of some basic constraint types to replaceability procedures.

In most tests, in addition to CNR algorithms, three other kinds of preprocessing were performed: simple arc consistency (AC), neighbourhood singleton arc consistency (NSAC), and neighbourhood inverse consistency (NIC). Arc consistency establishes consistency with respect to individual constraints; it guarantees that for any value v_i belonging to variable X_i and for any constraint C_k that includes X_i in its scope, that there is at least one tuple R_k that supports v_i (and this tuple is composed of currently viable values). NSAC is a form of arc consistency where the domain of X_i is first reduced to a single value v_i, and then the subgraph formed by X_i and its neighbours is made arc consistent. If this fails, then v_i is not neighbourhood singleton arc consistent and can be discarded. (See (Wallace, 2015) for more details.) NIC is another form of consistency, where for each v_i there is at least one neighbourhood solution that contains it. (See (Freuder & Elfe, 1996) for more details.) In the present work, algorithms for establishing these properties along with the replaceability algorithms formed a preprocessing step prior to carrying out the search for a solution.

The comparisons of greatest interest in this work were preprocessing runtimes and number of values removed. Runtime is a measure of efficiency, while number of values removed is the basic measure of effectiveness. In this work we also naturally considered the time to search for a solution following preprocessing, although as noted above this was subordinate to the main goals of these experimental tests.

In these experiments search was done with MAC-3 (or MGAC-3) with d-way branching. For smaller problems the domain/forward degree variable ordering heuristic was used, while for larger problems the domain/weighted degree heuristic of (Boussemart, Hemery, Lecoutre, & Sais, 2004) was used. All algorithms were coded in Lisp, and the Xlisp system was used for the experiments. Experiments were run under a Unix OS on a Dell Poweredge 4600 machine (1.8 GHz).

7.2. A note on replaceability with random problems

Simple probabilistic arguments show that very few values in a homogeneous random problem are likely to be replaceable, except for variables of very low degree. Interestingly, however, if inhomogeneities are introduced (of a kind to be described), then a small but significant proportion of the values in the problem are replaceable. Moreover, the average degree of variables with affected domains no longer remains close to 2 as problem density increases.

The analytic argument is straightforward. Given a consistent neighbourhood solution s_n with value v_f belonging to variable X_f (f is for 'focal'), we seek the probability

Table 2. Consistent, Replaceable Values Found in Homogeneous Random Problems of Varying Density[a]

	density		
	.10	.15	.17
mean values removed per problem	43.7	3.9	0.6
# problems with removals	25	14	2
mean degree of variables affected	1.86	2.12	2.00

[a]Data for sets of twenty-five problems.

of a value $v_{f'} \neq v_f$ such that $v_{f'}$ is consistent with the other assignments of s_n. For fixed tightness, this probability is equal to $(1 - p_2)$, where p_2 is the expected tightness of a constraint (i.e. the proportion of disallowed tuples, cf. (Gomes & Walsh, 2006)), raised to a power equal to the number of neighbouring assignments C, i.e. $(1 - p_2)^C$. (Here, we assume that p_2 is the same for all constraints, and that there is no distinction between values with respect to expected support. Note also that constraints between neighbouring variables must be satisfied by the original neighbourhood solution, whose values are the same for the alternative assignment.) Then, given a domain size d, the expected number of values that can replace v_f is $(d - 1) \times (1 - p_2)^C$.

For example, suppose $d - 1 = 10$ and $p_2 = 0.369$, so $(1 - p_2) = 0.631$, i.e. the constraints are fairly loose. Then we have the following estimates for the probability that $v_{f'}$ is a viable assignment, along with the expected number of values that can replace v_f (in this case equal to the value in the second column times 10):

#neighbours	$(1 - p_2)^C$	#replacements
1	.631	6
2	.398	4
3	.251	2.5
4	.159	2
6	.063	0.6
8	.025	0.2
10	.010	0.1
12	.004	0.04

In this case, there isn't much chance of finding a fully replaceable value when the number of neighbours exceeds 3 or 4.

This can be demonstrated using homogeneous random problems of 50 variables with domain size 15 and graph densities 0.10, 0.15 and 0.17 (a range of densities sufficient to demonstrate the effect). All constraints had an expected tightness of 0.45, but actual values could vary around this expectation. Problems were generated so as to be fully connected by first generating a spanning tree and then adding extra edges. Twenty-five problems of each type were tested; all problems had solutions. The results are shown in Table 2.

It can be observed that for problems with very sparse constraint graphs there are many replaceable values, but this is because there are many variables of degree two. (Occasionally, a single value was deleted from a variable of degree three.) Since low-degree variables are increasingly scarce as density increases, the number of values that are neighbourhood replaceable falls off rapidly. Moreover, since values that can be removed will be found in domains of variables of very low degree, their removal should have a negligible effect on the efficiency of obtaining a solution with any reasonable variable ordering, since these variables will be selected late in the search order.

However, if we consider other kinds of problems, a different picture emerges. Recall that the standard random model is extremely homogeneous with respect to its pa-

Table 3. Consistent, Replaceable Values in Homogeneous Random and Geometric Problems with Varying Tightness and Expected Degree[a]

	problems			
	random		geometric	
	2-60/8-40	2-70/8,9-15	all .45	2-60/8-40
mean values removed	3.7	5.7	6.8	27.6
# problems with removals	15	21	17	25
mn degree affected variables	3.28	3.56	5.74	6.59

[a]Column headers show support and proportion with this support. Thus, '2-60/8-40' means tightness of 0.2 with prob=0.6 and tightness of 0.8 with prob=0.4

rameter values. In particular, for each value in each domain the likelihood of support across each adjoining constraint is identical. (This is one of many reasons for considering these problems unrealistic.) However, problems can be generated based on random operations where the probability of support varies, either for a given constraint or for all constraints that a given value participates in.

In the present work this was done using a two-step process: given a set of probability values representing the possible likelihood of support, select one with a certain probability, and then select supporting values in the adjacent domain according to the probability selected. In the version of this scheme used in the present work, selection of a probability of support is done independently for each value and for each constraint.

In addition, random problems can be generated in which the constraint graph is nonhomogeneous, for example the so-called 'geometric problems' (Johnson, Aragon, McGeoch, & Shevron, 1991). These problems are generated by selecting points at random within the unit square to represent variables, and adding constraints between those whose Euclidean distance is less than some criterion, called the 'distance'. In our test case, the distance was 0.275. In addition, if there is more than one connected component, separate components are connected via the two variables with the smallest Euclidean distance between their points in the unit square, to make a single connected graph. Twenty-five problems were generated with the same number of variables, domain size and expected tightness as the random problems just described, and were filtered to select those with a number of constraints close to that for the problem set with density = 0.17, specifically, the expected number $\pm k$, where $k = 5$.

Results are shown in the Table 3. Clearly, irregularities of both kinds (constraint likelihood and support) are associated with more neighbourhood replaceable values. In addition, such values are no longer necessarily associated with variables of low degree. (Compare these results with the last column of the previous table.)

Although 28 out of 750 values is still a small proportion, it is interesting to find that under the rigorous conditions of selection (i.e. being replaceable with respect to all neighbourhood instantiations) one can find a number of such values. The reason for this, as well as the relatively high number of affected problems, is related to the 'clumpiness' of the geometric constraint graph. In these problems, the neighbourhood of a variable of high degree is highly constrained, so that a given value occurs in relatively few neighbourhood solutions. At the same time, there are more variables of low degree than in homogeneous random problems of the same density. The latter are mainly responsible for the large number of values removed overall, while the constrainedness of the neighbourhood subgraph results in an occasional value being replaceable even for variables of relatively high degree. (Note that the mean degree of affected variables measure in Table 3 does not consider the number of values removed for a given variable, but only whether or not one or more values are removed.)

For these reasons, experiments with random unstructured problems will be re-

Table 4. Values Removed and Run Times for Different CNR Algorithms[a]

	rem	time	nodes
AC	16	.00	21
NSAC	34	.03	20
NIC	34	.06	20
CNR-1	122	.16	20
CNRQ	122	.12	20
CNR-LY	122	30382	20

[a]Means for 50 problems: search nodes, total time (sec), and values removed during preprocessing (rem).

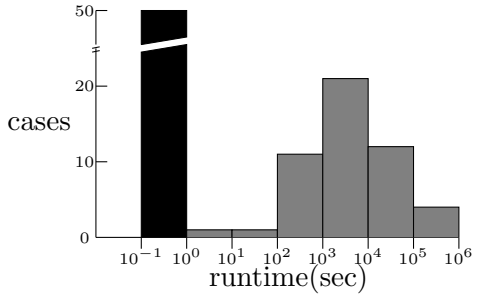


Figure 9. Frequency histogram for CNRQ (black) and CNR-LY (grey). Fifty problems in all. (Note log scale on abscissa.)

stricted to geometric CSPs whose values have variable support across constraints, as described above.

8. Comparing CNR Algorithms

8.1. Counting versus all-solutions search

For reasons that will become evident, this set of experiments was run with very small problems. The present problems were random geometric problems with 20 variables, domain size ten, and a distance factor of 0.200. Two levels of support were used: for each domain 60% of the values were consistent with 0.8 of the values in a given adjacent domain, while the other 40% were consistent with 0.2. Problems were filtered during generation so that, (i) all problems had 36 ± 2 constraints, (ii) all problems had solutions.

In these experiments search was done with MAC-3 using the domain/forward degree variable ordering heuristic. For all CNR algorithms, the queue order was lexical. (Unless otherwise stated, this is true for all subsequent experiments.)

Summary results including mean runtimes are shown in Table 4, both for CNR and for various consistency algorithms. Clearly, CNR algorithms remove many more values, although with these small problems search is minimal even following simple arc consistency. More important for present purposes is the enormous difference, amounting to five orders of magnitude, between the mean runtimes for the CNR algorithms based on all-solutions search and the solution counting algorithm, CNR-LY.

Further insight into this difference is obtained by considering individual runtimes, which are summarized in the frequency histogram shown in Figure 9. To accommodate the large variation among runtimes in a single figure, these times (in seconds) are binned according to successive powers of ten. As shown in the figure, when CNRQ is used, all 50 problems give runtimes less than one second, while for CNR-LY, runtimes are at least a second and sometimes as much as 10^5 seconds.

Table 5. Efficiency and Values Removed for Geometric Problems with Variable Support (120-Variable Problems)[a]

	rem[b]	proved	nodes	preproc	search
			satisfiable		
AC	39	–	46,265	0.1	266.2
NSAC	184	–	20,906	8.8	89.8
NIC	301	–	268	36.9	0.7
CNR-1	370	–	269	149.6	0.7
CNRQ	370	–	269	67.8	0.6
			unsatisfiable		
AC	40	0	93,297	0.1	536.2
NSAC	166	74	30,082	7.6	183.4
NIC	360	96	10,012	10.7	50.8
CNR-1	425	96	10,012	26.7	50.5
CNRQ	425	96	10,012	21.0	49.9

[a]100 problems per set. Means for values removed during preprocessing (rem), search nodes, preprocessing and search times (sec). For unsatisfiable problems, number proven so during preprocessing. Search with minimum domain/forward-degree.

[b] For unsatisfiable problems, restricted to problems not proven so by any preprocessing algorithm.

The large difference in efficiency between algorithms based on all-solutions search and those based on counting and inference is not due to the asymptotic complexity of NIC; in practice NIC is surprisingly efficient, as indicated by runtimes in the following sections. Instead, it is related to requirements for testing replaceability. For algorithms that find and test solutions, a single solution can serve to show that a value is irreplaceable; but something like this is impossible with counting methods, which must always tally up solution counts in order to compare them with the bounds. With vast numbers of neighbourhood solutions (cf. Table 14), this can lead to the large differences found here.

Since this problem is not related to any peculiar feature of random problems, it will also manifest itself with CSPs with ordered domains and constraints. In view of this, and since the basic complexity is not improved by the counting procedures, in the rest of the paper only algorithms based on all-solutions search will be considered.

In this experiment CNR-1 and CNRQ had similar runtimes. But as one would expect from the complexity analysis, with larger problems a clear difference emerges, as shown in the next series of experiments.

8.2. CNR-1 versus CNRQ

We begin with another test using geometric problems. In this experiment, problems had 120 variables, domain size 20, and two levels of support: for 80% of the values tightness (obverse of support) was 0.3; for 20% it was 0.7. The (Euclidean) distance parameter was 0.17 and a target was set for 540 constraints (± 3), giving a graph density of 0.076. A large set of these problems was generated, and the first 100 problems with and the first 100 problems without solutions were used for testing.

Results are shown in Table 5. Here, we see that ten times as many values are removed by CNR algorithms as by AC. In addition, although NIC was also quite effective, with CNR even more values were removed during preprocessing. However, it is worth noting that for these problems NIC removed a sufficient number of values to make subsequent search as efficient as with the CNR algorithms. (Note that for the unsatisfiable set, mean search nodes is based on only four problems.)

Table 6. Efficiency and Values Removed for Distance Problems[a]

	rem[b]	proved	nodes	preproc	search
			satisfiable		
AC	0	–	105	0.0	0.2
NSAC	7	–	94	0.2	0.1
NIC	13	–	91	0.6	0.1
CNR-1	295	–	60	427.9	0.0
CNRQ	295	–	60	337.7	0.0
			unsatisfiable		
AC	0	0	171	0.0	0.5
NSAC	11	0	139	0.2	0.4
NIC	29	4	110	0.6	0.3
CNR-1	300	4	33	632.4	0.1
CNRQ	300	4	33	649.2	0.1

[a]50 problems per set. Means as in Table 5. Search with minimum domain/forward-degree.[b]See Table 5 note.

For these problems, CNRQ is much more efficient than CNR-1. With even larger problems of this type (not shown), CNRQ improves on CNR-1 by more than an order of magnitude. This corresponds to the difference in worst-case complexity.

In this case, when neighbourhood replaceable values are discarded before search, this has a significant effect on subsequent search effort. In some cases, the difference exceeds the difference in preprocessing time between arc consistency and CNR algorithms.

8.3. Results for different constraint types

The next three experiments attest to the superiority of CNRQ over CNR-1 for problems with different constraint types, although the size of the difference varies greatly. For this purpose, we used three types of random structured problems:

- Random distance problems, where all constraints were of the form $|X_i - X_j| > k$ or $|X_i - X_j| \geq k$. In words, the absolute difference between values assigned to adjacent variables has to either exceed or be equal or greater to some value k (which might or might not differ among constraints).
- Random relop problems where constraints between adjacent domains involved one of the six basic arithmetical relational operators. Thus, all constraints were of the form $X_i \otimes X_j$, where $\otimes$ is a relational operator. In the problems used in these experiments, half the constraints were of the type $\geq$ and half were $\neq$. The latter ensured that the problem class was intractable.
- Random k-colour problems where constraints are all inequality constraints. (In this form of colouring problem, the goal is to label nodes of a graph with colours drawn from the same set of k colours while satisfying the constraint that adjacent nodes must have different colours.).

In each case, the problems tested were close to the largest that could be solved by CNR algorithms in a reasonable amount of time.

Distance problems had 50 variables, domain size 10, constraint graph density 0.08, and fixed $k = 3$. Approximately half the constraints were of the form $|X_i - X_j| > 3$ and half were of the form $|X_i - X_j| \geq 3$. Relop problems had 40 variables, domain size 10, and constraint graph density 0.60. For each problem class, these parameter values allow one to readily generate problems with and without solutions.

Colouring problems had 100 variables, domain size 6 (six colours), and densities of 0.02 (tree-structured constraint graph), 0.05, 0.07, and 0.10. For each density 50

Table 7. Efficiency and Values Removed for Relop Problems[a]

	rem[b]	proved	nodes	preproc	search
			satisfiable		
AC	0	–	39,659	0.0	59.3
NSAC	260	–	160	2.6	0.2
NIC	269	–	189	27.9	0.2
CNR-1	276	–	181	1021.6	0.1
CNRQ	276	–	181	960.6	0.1
			unsatisfiable		
AC	0	0	22,140	0.0	46.5
NSAC	248	31	80	2.1	0.1
NIC	256	43	123	25.2	0.1
CNR-1	265	44	126	512.2	0.1
CNRQ	265	44	126	499.3	0.1

[a]50 problems per set. Means as in Table 5. Search with minimum domain/forward-degree.[b]See Table 5 note.

problems were generated. The reason for examining a series of densities is that for many problems of this type, especially if randomly generated, there is a relation between the domain size and degree of a variable that determines whether there are replaceabe values. This as indicated by the following proposition.

Proposition 9. Suppose a k-colouring problem is satisfiable and all domains are initially composed of the same k colours. If the neighbourhood of variable X_i can be coloured with up to r colours, where $r \leq k-1$, then some values in the domain of that variable will be neighbourhood replaceable iff the number of values (colours) in the domain of X_i exceeds the value of r by two or more.

Proof. If d, the size of the domain of X_i is $\leq r$, then each colour can be associated with a unique set of $d-1$ other colours in the neighbourhood, and no value is replaceable. If D_i has $r+1$ colours, then each colour can be associated with a unique set of r other colours in the neighbourhood, and no value is replaceable. However, if d is $\geq r+2$, then for any instantiation of the neighbours, there will be at least two colours that can be assigned to X_i to give a consistent subproblem. Hence, for any colour and any consistent neighbourhood assignment, there will be a replacement value. □

That some configurations do not allow $k-1$ different colours to be used even if the neighbourhood is $\geq k$ can be shown by example. Suppose each value in the neighbourhood of X_i is adjacent to all of the variables in a clique of size $k-1$, and there are no constraints between variables in the neighbourhood. In this case, all variables in the neighbourhood must be coloured with the same colour. In this case, some values in D_i will be replaceable if it is of size 3 or more. However, this will be a very rare configuration in randomly generated problems. With other less extreme examples the situation is less clear.

From this we conclude that, since k-colouring problems of this type generally have small domains, only very sparse problems are likely to have appreciable numbers of replaceable values.

For distance problems that were satisfiable (Table 6), no values were removed by AC and very few by NSAC or NIC; however, more than half were replaceable. For relop problems (Table 7), although there were no arc-inconsistent values, both NSAC and NIC were able to remove almost as many values as CNR. In this case all three algorithms led to a dramatic reduction in search effort, although, as with distance problems, runtimes for CNR algorithms were much greater.

For colouring problems (Table 8), large numbers of values were replaceable for problems of very low density. At the same time, none of the consistency algorithms was able to remove values. Since search never required more than 120 nodes on average,

Table 8. Efficiency and Values Removed for k-Colour Problems[a]

	.02		.05		.07		.10	
	rem	preproc	rem	preproc	rem	preproc	rem	preproc
AC	0	0.00	0	0.01	0	0.01	0	0.02
NSAC	0	0.23	0	0.27	0	0.30	0	0.38
NIC	0	0.46	0	1.20	0	1.75	0	2.71
CNR-1	413	3.42	86	12.94	26	15.85	3	13.07
CNRQ	413	2.91	86	9.87	26	9.86	3	6.89

[a]50 problems at each of four graph densities. 'rem' is values removed. 'preproc' is preprocessing time in sec.

these data are not shown in the table. (Reflecting differences in constraint graphs even when these were trees, the number of values removed by replaceability algorithms for problems in the 0.02 density set ranged between 367 and 466.)

These tests show that problems with distance, relop or inequality constraints are all amenable to simplification by removing replaceable values. Moreover, with relatively small problems this can often be done with the present algorithms in a fairly reasonable amount time.

8.4. Experiments with structured problems with global constraints

Experiments were also conducted using problems with different types of global constraint. For this purpose, we used the 71-variable driverlog problem from the well-known series. (Driverlog problems are a form of transportation problem in which both drivers and trucks are organized in order to deliver packages from locations a to locations b.) Global constraints were added to the basic binary problem using a random problem generator devised for this purpose. This enabled us to generate many individual problems according to various specifications. The present experiments are based on three classes of problem:

- Problems with two among constraints. (Among constraints require that some number of variables in the scope of the constraint must take a value from a specified set.)
- Problems with two max and two min constraints. (Max (min) constraints require that the maximum (minimum) value of all but one variable in the scope must be equal to the value of the remaining variable.)
- Problems with two among and two max constraints.

These constraints were chosen because they seemed consistent with the original driverlog representation.

In the first class of problems, global constraints were of arity 8-12. In addition, (1) the maximum number of variables subject to the among restriction (i.e. their values had to be from the designated set) was three-quarters of the variables in the scope, (2) the maximum number of values that could be included was all possible values in the domains of the variables in the scope, (3) the degree of overlap allowed between the scopes of the global constraints was zero. For each feature, the actual value was chosen randomly within the specified range for each problem and constraint. For example, the number of variables to be subjected to the domain value restriction was $k = 1$ + random(k_{max}), i.e. a value between 1 and k_{max}, the maximum number specified. Given the 3/4 specification, if the arity was 8, then k could be between 1 and 6.

In the second class of problems, global constraints of both types had arities in the range 8-12. The overlap between constraints of the same type was zero, while between

Table 9. Efficiency and Values Removed for Driverlog Problems with Added Global Constraints (Satisfiable Problems)[a]

	rem	nodes	preproc	search
		among constraints		
AC	19	3,071	0.0	1.4
NSAC	24	2,735	0.1	1.3
NIC	41	1,055	0.8	0.5
CNR-1	45	684	2.7	0.3
CNRQ	45	684	1.3	0.3
		max & min constraints		
AC	44	1,051	0.0	0.5
NSAC	54	397	0.1	0.2
NIC	55	364	0.9	0.2
CNR-1	61	177	3.5	0.1
CNRQ	61	177	1.8	0.1
		among & max constraints		
AC	40	1,838	0.0	0.9
NSAC	45	1,411	0.1	0.7
NIC	70	261	0.8	0.1
CNR-1	74	175	2.1	0.1
CNRQ	74	175	1.1	0.1

[a]100 problems per set. See Table 5 notes. Search heuristic is min domain/forward degree.

different types it was allowed to vary between 0 and 50% of the smaller scope.

In the third class of problems, among constraints had arities between 4 and 6, while max constraints had arities between 8 and 12. The other among features were the same as in the first problem class. In this case, no overlap was allowed between the scopes of any of the constraints.

In examining problems generated in this way, we found cases where a randomly generated constraint wasn't reasonable, e.g. an among constraint where only one value was to be used for several variables. Partly to avoid these cases, and since it wasn't always clear from inspection which constraints 'made sense', we simply used problem difficulty based on MGAC as a criterion for inclusion in the test set. To this end, a filtering criterion of n search nodes was used: for the first and third classes of problems it was 500 for problems with solutions and 1000 for unsatisfiable problems. Because of the greater difficulty in finding any difficult problem in the second class, these values were reduced to 250 and 150, respectively. (In this class unlike the other two, acceptable unsatisfiable problems were much rarer than acceptable satisfiable problems.) The final problem set size was 100 for satisfiable and unsatisfiable problems in classes one and three, and satisfiable problems in the second class. Thirty unsatisfiable problems were obtained for the second problem class after generating 50K problems; this was deemed sufficient for the present tests.

The results of these experiments are shown in Tables 9 and 10. Although these problems are relatively easy, they show that the usual ordering of value deletions, with CNR > NIC > NSAC > AC, holds across different types of global constraints. For these problems, regardless of the specific class, CNRQ was consistently better than CNR-1; for satisfiable problems this was by a factor of 2.

It is also worth noting that NIC often showed a marked advantage over NSAC on these problems, and NSAC was usually but not always more effective than AC. For some classes of problems without solutions, NIC was able to prove unsatisfiability for most or all of the problems.

To summarize: in line with complexity results, CNRQ was found superior to CNR-1 (and CNR-LY) on every class of problems tested, although in some cases the difference was surprisingly small. Therefore, in the rest of the paper we will only consider the

Table 10. Efficiency and Values Removed for Driverlog Problems with Added Global Constraints (Unsatisfiable Problems)[a]

	proved	rem[b]	nodes	preproc	search
		among constraints			
AC	0	18	16,948	0.1	8.0
NSAC	0	22	15,230	0.8	7.1
NIC	96	25	127	0.2	0.1
CNR-1	96	29	180	0.3	0.1
CNRQ	96	29	180	0.2	0.1
		max & min constraints			
AC	0	42	3,421	0.0	1.8
NSAC	23	54	402	0.1	0.2
NIC	23	55	398	0.6	0.2
CNR-1	23	59	389	1.3	0.2
CNRQ	23	59	389	0.9	0.2
		among & max constraints			
AC	0	39	5,284	0.0	2.6
NSAC	0	44	4,682	0.1	2.4
NIC	100	–	0	0.2	0.0
CNR-1	100	–	0	0.2	0.0
CNRQ	100	–	0	0.2	0.0

[a]100 problems per set except for max/min. [b]See Table 5 notes.

CNRQ algorithm.

9. Inferring Replaceability: Theoretical Analysis

9.1. Inference methods

Many kinds of intensional constraints afford some basis for inferring whether or not a given value is replaceable. We consider negative cases first, cases where we should avoid checking individual values. The most obvious involves the equality constraint. If there is a binary *equality* constraint between X_i and X_j, then any value a of variable X_i that satisfies this constraint will be associated with a unique value a in the domain of X_j, and any other value in the domain of X_i will not support a in X_j. Hence, no other value can replace it. A similar argument can be made in cases where the difference between two values must satisfy an equality constraint, as with radio link frequency assignment problems (RLFAPs). For example, if $|a - b| = k$ satisfies an absolute difference constraint between X_i and X_j, then depending on whether a is greater or smaller than b, only $b + k$ or $b - k$ may be substituted for a, respectively (assuming all values are nonnegative), and in this case, of course, $b+k$ or $b-k$ is equal to a itself.

And, as we have already seen (cf. Proposition 9), it is possible to state some restrictions on neighbourhood replaceability for k-colouring problems. Namely, that if the neighbourhood of a variable can be assigned a number of colours greater than or equal to the number of available colours minus one, then that variable will have no neighbourhood replaceable values.

For problems with distance constraints of the form $|X_i - X_j| > k$ or $|X_i - X_j| \geq k$, there are several ways to avoid checking individual values. First, depending on the minimal distance and the range of domain values, it may be possible to infer replaceability for values near the extremes, since support for extreme values will always include the support for somewhat less extreme values. More precisely,

Proposition 10. We assume distance constraints of the form $|X_i - X_j| > k_{ij}$, where X_i

and X_j are adjacent variables. We also assume that the original domains are composed of values $1 \ldots d$. (Without loss of generality, we assume that d is the same for all domains.) Then for sets $D_l = l \ldots 1 + k_{min}$ and $D_u = d - k_{min} \ldots u$, where l and u are the smallest and largest members of D that are still viable and k_{min} is the smallest k for all constraints adjacent to X_i, l and u can replace any other member in D_l and D_u, respectively.

Proof. For set $l \ldots 1 + k_{min}$ in the domain of X_i, any value $v_j \in X_j$ must be greater than k_{min} to satisfy the $>$ constraint. In this case, if $v_j - v_i$, where $v_i \in D_l$, satisfies a constraint, then $v_j - l$ will also satisfy it. A similar argument holds for u and D_u, except in this case v_j must be less than $d - k_{min}$. □

Obviously, a similar proposition holds for distance constraints where the relation is $\geq$; in this case k_{min} must be used instead of $k_{min} + 1$ for the smallest values and $k_{min} - 1$ instead of k_{min} for the largest values. These rules will also work for mixed $>$ and $\geq$.

A second form of inference is based on symmetry relations with respect to support for values in the lower and upper halves of the range such that if a value in one half-range can be shown to be replaceable then its complement in the other half-range is also replaceable. However, there are conditions to be met to ensure that this manoeuvre is valid. To show this, we first consider the simplest case in which all domains contain a complete sequence of integers, i.e. there are no breaks in the sequence, and constraints are all of the form $|X_i - X_j| > k$, i.e. there is a single k value. (In the discussion that follows, we also assume that k is always positive; this condition holds for the problems we will consider.)

Without loss of generality, we assume that all domains consist of all integer values between 1 and some n. In what follows, we define the midpoint of such a sequence as equal to $\lceil \frac{n}{2} \rceil$ for odd n and a value half-way between $\frac{n}{2}$ and $\frac{n}{2} + 1$ for even n.

Definition 15. Two values v_i and v_j are *midpoint symmetric* with respect to the midpoint of a range of integer values from 1 to n if they are equidistant from the midpoint.

Proposition 11. Suppose that each domain in problem P is a range of ordered values from 1 to n. Then, for problems based on distance constraints, if a value v in D_i is replaceable, then its midpoint-symmetric value, v' is also replaceable, with one caveat: that v' is not part of the replacement set. More generally, this symmetry relation holds for the domain D_i of variable X_i, provided there is a symmetrical set of values around the midpoint of D_i and this condition also holds for each of its neighbours.

Proof. Without loss of generality, we assume that the initial domains are complete integer sequences between 1 and n_i for each variable X_i. (The necessary starting point for the proof is that for each value v below the midpoint, there is a value $v' = n_i - v + 1$ above the midpoint.) From here on, we drop the subscript and refer to endpoint n. Suppose we have found the first replaceable value v for a given variable X_i. We consider any complete set of replacement values, R_v, that does not include v'. Since the domain is a midpoint-symmetrical sequence, for every replacement value w, where $w - v = d$, there will be a corresponding domain value w' where the difference between w' and v' is also d.

Given this, we also know that for every neighbourhood tuple where w can replace v, this is true by virtue of the fact that for each constraint with X_i where $|v_i - v_j| > k$, then $|w_i - v_j| > k$. In addition, for every tuple consistent with v_i and also with a replacement value w_i, there will be a corresponding tuple consistent with v_i' and w_i', since the domains of the neighbouring variables are also midpoint symmetric.

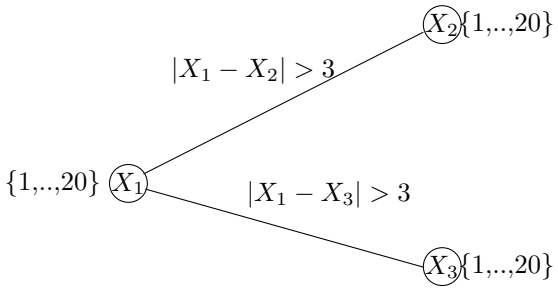


Figure 10. Example to illustrate inference based on midpoint symmetry for distance constraints.

Hence, for this case $|w'_i - v'_j| > k$. Since the same relations hold for every value in the replacement set $\{w_i\}$, then $n - v + 1$ can also be replaced.

Next, we consider the set of neighbourhood tuples supported by either v or $n - v + 1$, respectively, at some later point in the process. At this point, the cardinality of these sets may differ depending on the current state of neighbouring domains. However, taking either set and some replacement set $\{w\}$ or $\{w'\}$, the complementary set, $\{w_{comp}\}$ or $\{w'_{comp}\}$, will be found in the domain of X_i by virtue of the symmetry of the domain around the midpoint and the midpoint symmetry of the adjacent domains. Hence, if v is replaceable then so is its midpoint symmetrical value v'. □

The following example illustrates these arguments (cf. Figure 10). Here, we consider a variable X_1 with domain $\{1, \ldots, 20\}$ and two midpoint symmetrical values, 4 and 17. In this case, 4 is consistent with values 8,..., 20 in the domains of X_2 and X_3, while 17 is consistent with values 1,..., 13. We suppose that 4 has been found to be replaceable and the replacement set R_v is $\{8, 20\}$. In this case, the corresponding set $\{1, 13\}$ has the same distance relations to 17 as R_v has to 4. And since the adjacent domains are also midpoint symmetric, it will have the same distance relations to the values consistent with 17 as R_v has to 4. However, if the adjacent domains are not midpoint symmetric, we cannot assume that if R_v is a replacement set for 4, then the corresponding set R'_v is present, allowing 17 to be removed without directly testing for replaceability.

These arguments can be extended to constraints of the form $|X_i - X_j| < k$, or to corresponding forms where the operator is $\geq$ or $\leq$. In addition, for these problems, if a value is filtered by an inconsistency algorithm, then its symmetric complement will also be deleted. (Although AC algorithms are not able to delete any values, SAC-based reasoning (with neighbourhood SAC or higher-order SAC-based methods) is effective.) This ensures that asymmetries are not introduced by inconsistency processing.

The arguments can also be extended to problems with certain kinds of global constraints (that are used in the following section on experiments). These are k-ary constraints based on the maximum or minimum distance between all pairs of variables within the scope of the constraint. In this case, the arguments based on symmetry around the midpoint of the domain also apply. Consider, for example, the minimum distance greater than k_{min} constraint. This is basically a conjunctive relation, so for any variable X_i in such a constraint all distances between the value assigned to X_i and values in other variables in the constraint must be greater than k_{min}. But in this case, if the requisite conditions hold, then Proposition 10 applies for values v in domain D_i. This generalisation to k-ary constraints is not affected by distance relations between pairs of neighbours of X_i, since they remain the same following a valid replacement.

The same arguments can be made for maximum distance less than k_{max} constraints, which have the same conjunctive character.

It is of particular interest that with some elaborations these arguments can also be applied to problems with disjunctive constraints of the same character, which includes scheduling problems. Here, the constraints usually have the form $X_i + k_1 < X_j \bigvee X_j + k_2 < X_i$. (In this discussion we assume that all k_x are non-negative integers.) The two components can be rewritten as $X_j - X_i > k_1$ and $X_i - X_j > k_2$, respectively. This, in turn, implies the constraint $|X_i - X_j| > \min(k_1, k_2)$. This suggests, but does not prove, that the same kind of midpoint symmetrical reasoning is possible for problems with these constraints, since values that can replace a given value must satisfy the original disjunct.

Proposition 12. For problems with disjunctive constraints of the form just described, if value v is replaceable, then, provided (i) the domain is symmetric around its midpoint, (ii) its midpoint symmetric value v' is consistent and not part of the replacement set for v, (iii) each adjacent domain is also symmetric around its respective midpoints, v' is also replaceable.

Proof. Given these three conditions, if a replacement set R_v is found for value v (because they satisfy the same disjunctive constraints), then there will be a corresponding set R'_v that has the same distance relations to v' as the elements of R_v have to v. Also, because of condition (iii), for each adjacent constraint, v' will satisfy the same disjunct as v for any given adjacent value that supports v. But, since the distance relations between the elements of R'_v and the support sets of v' in adjacent domains are the same as those between elements of R_v and the support sets for v, then R'_v can serve as the replacement set for v'. □

Turning to the binary relop problems described earlier and excluding equality constraints, we can also make potentially significant inferences. In what follows, we assume that the initial domains are complete ranges, although they do not have to be identical. Under this condition, it can be shown, within certain limits, that when such problems are reduced by neighbourhood replaceability algorithms, the remaining values in any domain form a single complete subrange.

Conjecture 1. We consider binary CSPs based on arithmetic relational operators other than equality constraints (i.e. $>, <, \geq, \leq$ and $\neq$), where the relations are between single values (e.g. $X_i \neq X_j$), where the original domains are complete integer sequences, and where there are no inequality constraints associated with singleton domains either initially or after consistency testing. In addition, we assume that values in a domain are tested in lexical order. Under these conditions, the set of irreplaceable values produced by a CNR algorithm is always a complete subrange.

It can be shown by example that some value orderings can lead to sets of irreplaceable values that do not form a complete subrange, although for the problems tested, this is surprisingly rare. Thus, if values are tested in the order, lowest, highest, followed by a binary-search-like order in which values are taken from the middle of the domain, then the middle of each subdomain above and below the last value chosen, etc., then it has been found that for a few domains in some problems tested, there are gaps in the sequence of irreplaceable values ordered by numerical value. (The problems were those used in Section 10.3.)

At this time, this Conjecture in its general form has not been proven because the existing proof strategy requires that a large number of cases be tested. This can only be done for domains of limited size and a limited number of inequality constraints. The present proof allows us state a restricted form of the conjecture as a proposition:

Proposition 13. Conjecture 1 holds for binary CSPs with domains up to size eight and with up to four inequality constraints.

Because the argument behind this proposition is involved and because these are rather specialized problems, the proof along with subsidiary lemmas has been relegated to an appendix. In addition to the computational work described there, Conjecture 1 has been tested on billions of instances with large enough domain sizes and numbers of inequality constraints to cover all of the problems used in our experiments. Given these results, we will tentatively assume that Conjecture 1 applies to all these problems.

Given that Conjecture 1 holds, we can make two further significant inferences:

Proposition 14. Where Conjecture 1 holds, if a value v in the domain of X_i is neighbourhood replaceable, then either every value greater than v is neighbourhood replaceable or every value less than v is neighbourhood replaceable.

Proof. Suppose there were irreplaceable values greater than v and less than v. Since v is neighbourhood replaceable, the resulting set of irreplaceable values would not form a complete subrange. □

Proposition 15. If Conjecture 1 holds, then for each problem P having the requisite properties, there exists a reduced set of irreplaceable values in each domain that forms a complete subsequence.

Proof. It was demonstrated earlier that replaceability algorithms do not have unique fixpoints. Therefore, regardless of whether there are sets of irreplaceable values in a given domain of P that do not form complete subsequences, if Conjecture 1 holds, it will always be possible to perform CNR using a lexical value ordering to produce a set of irreplaceable values with this property. □

9.2. Ordering heuristics for CNR search algorithms

In addition to using inferences to avoid search, we can also organize search so as to reduce the number of neighbourhood solutions that need to be processed when considering a given variable. To do this we introduce some heuristics for ordering the queue of variables. These are based on the observation that the efficiency of the CNR algorithm can be improved if the neighbours of a variable with a reduced domain are already on the queue and therefore do not need to be added back. This can be done if more values are removed sooner during the procedure. In addition, it is obvious that the effort required for an all-solutions search may be reduced if the neighbouring domains have already been reduced in size.

Two queue ordering heuristics were tested with these goals in mind. The first simply orders the queue by ascending degree of the variables. The second orders the queue by ascending (degree - width), where the width is the number of constraints with variables whose replaceable values would have already been deleted because they appeared earlier in the queue.

Both heuristics can be extended to problems with k-ary constraints. For the first heuristic, this is straightforward. For the second, we adopted the following procedure. For each variable X_i, the total number of adjacent variables was counted, including duplications of variables that appeared in more than one constraint. Then if an adjacent variable was already in the queue, the total was decremented according to the number of adjacent constraints it appeared in.

Since the degree-minus-width heuristic proved to be appreciably better than the other, this is the only one reported in the following section.

10. Inferring Replaceability: Experimental results

10.1. Distance problems

The ideas presented in the previous subsection permit various kinds of strategies for inferring replaceability. First, consider the finding that values near the extremes can be discarded. To take advantage of this, two schemes were tried. The first was simply to order the values for testing from the middle value in the domain outward to the extremes; this guarantees that no extreme values are tested before the ones nearest to them. A more sophisticated strategy is to test the extremes for consistency, and if they are consistent, to discard values where the difference from the extreme value is less than or equal to the value of k. (This is for $> k$ constraints; for $\geq$ constraints, the difference from the extreme value must be less than k.) [1]

With some problems it is possible to avoid checking the subranges of values near the extremes in the domains of adjacent neighbourhood variables during neighbourhood search. The reasoning here is that any solution that includes those values will eventually be lost when those values are discarded later during preprocessing. If, on the other hand, an extreme value is lost due to inconsistency, then the adjacent variables will be rechecked, and in this case the condition can be flagged so that values near the missing extreme are included in the search.

The symmetry between values on either side of the midpoint was taken advantage of in a straightforward way: given that the conditions specified in Proposition 11 held, then removal of a replaceable value was attended by the removal of its symmetric counterpart.

For the main tests, problems were used that were somewhat more difficult than those used for the basic results, while still being small enough to allow the basic CNR algorithm to be run to completion on all problems. Problems had 30 variables, domain size 15, graph density 0.25. All constraints were of the form $|X_i - X_j| > 3$. These problems were culled from a set of 1000; they were the first 50 problems that required more than 500 search nodes to solve and were also satisfiable.

The results of this experiment are shown in Table 11. In the table, 'queord' and 'que' refer to the minimum degree minus width ordering already described, 'subrngx' and 'subx' refer to exclusion of values near the extremes, 'subrngxx' and 'subxx' to the additional exclusion of values in adjacent domains during neighbourhood search.

In the first place, these results show once again that replaceability algorithms remove many more values than local consistency algorithms. At the same time, even with problems of this size the time required to do this is exorbitant. However, the ordering and inference strategies devised for these problems allow dramatic improvements in efficiency, to the extent that replaceability methods become almost as efficient as an algorithm like NIC.

Perusal of the table indicates that every strategy improves efficiency. However, the degree of improvement is vastly different for different strategies. Since the value ordering had only a small effect (about a 25 percent improvement in run time), it was not used in any further test. Both queue ordering and symmetry had somewhat greater effects. In contrast, the subrange strategy had a marked effect on performance, reducing run times by three orders of magnitude. Moreover, it could be combined with any of the other strategies to reduce run times even further.

For testing these methods on problems with max-less or min-greater global con-

[1] In the present code, the set of adjacent constraints was always checked to find the minimum k, although for the problems tested k was constant.

Table 11. Efficiency and Values Removed for Problems with Distance Constraints Using Inference and Ordering Strategies[a]

	rem	nodes	total time
AC	0	4233	1.3
NSAC	0	4233	1.5
NIC	33	3084	1.6
CNR(Q)	330	53	16,092.0
CNR-queord	330	53	4,381.2
CNR-valord	330	53	12,792.6
CNR-val+que	330	53	3,360.9
CNR-subrngx	330	53	19.6
CNR-subrngx+que	330	53	14.6
CNR-subrngxx	330	53	19.8
CNR-subrngxx+que	330	53	11.9
CNR-sym	330	53	5,423.5
CNR-sym+que	330	53	1,558.2
CNR-subxsym	330	53	13.6
CNR-subxsym+que	330	53	12.3
CNR-subxxsym	330	53	15.1
CNR-subxxsym+que	330	53	7.7

[a]Means for 50 problems. Times in sec. 'rem' is values removed. 'nodes' is search nodes using min domain/forward degree.

straints (described in Section 9.1), 20-variable problems were used. The domain size was eight, and the mean degree was two. Constraint arities were 2, 3 and 4 in the following proportions: 0.3, 0.5 and 0.2. (Constraint frequencies were first calculated on this basis and then adjusted slightly during problem generation in order to achieve the designated mean degree exactly, since this was the parameter that had been specified.) For arity 2, the relations used were $>$ and $\geq$ in equal proportion. For arity 3, the ratio of max-less to min-greater was 4:1; for arity 4, all constraints were max-less (cf. Section 9.2). Fixed values of k_{min} and k_{max} were used in these constraints. For arity 2, the value was 2; for arity 3 and 4, it was 6 for max-less and 2 for min-greater.

These parameters gave problems that varied widely in difficulty for minimum domain over forward degree, although most were trivially solvable ($\leq$ 20 search nodes). Only one percent of the problems generated were unsatisfiable. To obtain samples for testing, a set of 8000 problems was generated. For satisfiable problems, trivial problems were avoided; to this end a minimum number of search nodes of 100 was used, and the first 50 problems generated that met this criteria were selected. For unsatisfiable problems, since the focus of interest was on comparing problems proved unsatisfiable during preprocessing, the first 50 problems were selected without any filtering (other than the fact that they had no solutions). Despite this relaxation of restrictions, problem difficulty varied widely across the sample.

For brevity, only the set of problems with solutions is shown here. (Improvements with these inference and heuristic techniques were also found for problems without solutions, but since the original results were largely due to NIC proving unsatisfiability, these differences are less significant.) Results are shown in Table 12.

Interestingly, for these problems, using the queue ordering heuristic resulted in orders-of-magnitude improvements, while the subrange or symmetry inferences either had no effect or produced worse performance (although when subrange reasoning was extended to neighbouring variables during the all solutions search, there was a distinct improvement). As with the binary problems, combining these techniques led to even better performance.

Given these results, it is of interest to see how efficient the best combination is with larger problems. Two sets of problems were tested with the most efficient combination

Table 12. Efficiency and Values Removed for Problems with N-ary Distance Constraints Using Inference and Ordering Strategies[a]

	total time	failed
CNR(Q)	>16.7K	1
CNR-queord	17.5	0
CNR-subrngx	>27.0K	2
CNR-subrngx+que	106.0	0
CNR-subrngxx	3,225.9	0
CNR-subrngxx+que	12.1	0
CNR-sym	>16.6K	1
CNR-sym+que	17.4	0
CNR-subxsym	941.1	0
CNR-subxsym+que	10.4	0
CNR-subxxsym	1,846.5	0
CNR-subxxsym+que	2.5	0

[a]Means for 50 problems. Times in sec. 'failed' means stopped before completion. Other notes as in Table 11.

(shown in the last row of Table 12). The first were 40-variable problems with the same features as the 20-variable problems, except that the mean degree was set to 2.5 instead of 2, which resulted in more difficult problems. CNRQ with the combination of advanced features was able to finish these problems in an average of 108 seconds. Problems in the second set had 30 variables, domain size 10, mean degree of 3 instead of 2; other features were like the original 20-variable problems. In this case the mean was 315 seconds. In both cases, almost no variable assignments needed to be retracted in the subsequent search for one solution. Hence, despite the large overhead incurred by replaceability algorithms, inference methods together with queue ordering heuristics make it possible to handle problems of moderate size fairly efficiently.

10.2. Scheduling problems

Similar tests of inferences were also made on scheduling problems using the Taillard-4-100 series of the Taillard open shop scheduling problems (Taillard, 1993). In the present implementation of symmetry inference for scheduling problems, if a particular value was replaced, the midpoint symmetrical value was the next one tested. If it was consistent, then it was replaced; otherwise, a symmetry flag for that domain was set to nil, and after that no inferences based on symmetrical values were attempted for that domain.

The basic CNRQ algorithm required an average of 30,000 seconds on these problems. Using either subset or symmetry inferences alone reduced the mean to 16 and 14 thousand seconds, respectively. Using these methods together along with skipping the extreme values brought a further reduction to 8 thousand seconds. Moreover, for some problems it was possible to reduce the time from 10^4 to 10^2 or 10^1 seconds.

So, while not finding the spectacular overall reduction obtained for simple distance problems, it was possible to bring about a clear reduction with these methods, which was marked for some problems. Using a strategy of capping algorithm effort, it would therefore be possible to simplify an appreciable number of problems, although they would still have to be rather small.

10.3. Relop problems

Devising procedures to take advantage of the subrange property specified in Conjecture 1 and Propositions 13-15 is not straightforward. In the first place, even though we are assured that there exists a complete subrange of irreplaceable values, information regarding the replaceability of a particular value is not sufficient for useful inferences, i.e. those resulting in the immediate removal of values other than the one just tested. Even with two or more values, the reasoning is complex and depends on assessing a number of cases. But perhaps the most important problem is that a value that is irreplaceable at some point in the process may eventually become replaceable. Hence, if a value is determined to be irreplaceable, this must be considered as tentative until the replaceability algorithm has terminated.

These difficulties were handled in the following way. The procedure made use of two specialized data structures. The first, called the domain map, kept information about the current status of each value. For each value, a single integer indicated whether it was certainly replaceable (-2), possibly replaceable (-1), possibly irreplaceable (1), or not yet tested (0). In addition, while the values of a domain were being tested, rather than a simple list of values a list of subdomains was maintained. Associated with each entry were two flags that indicated whether the value just beyond each end of the subdomain was replaceable, potentially irreplaceable, or not yet tested.

The replaceability procedure began as before, except that instead of testing values in lexical order, first the end values were tested, and after that, each value chosen was the middle value in the current subdomain. Once both flags had been set to either T or F, indicating a (potentially) irreplaceable or replaceable value, respectively, had been found, then inferences could be made to avoid testing some values. The basic inferences involved the following conditions (here, testing the domain of the current variable is called a "step"). First, we consider cases where the value just tested is replaceable.

- Both flags F and at least one value was previously identified as irreplaceable. In this case each value in the subdomain is labeled with -1, and the subdomain is not tested further during this step.
- Otherwise if both flags are F, the domain is simply divided into two subdomains, which are put back on the queue.
- Flags T and F. In this case values in the half-subdomain between the current value and the value at the 'F-end' are labeled with -1 and not tested during this step.
- Both flags T. In this case the domain is divided into two subdomains, which are put back on the queue.

If the value just tested was found to be irreplaceable (at this point), then we consider these cases:

- Both flags T. Values in the subdomain are labeled 1 (tentatively irreplaceable) and the subdomain is not tested further during this step.
- Flags T and F. Values between the current value and the 'T-end' are labeled 1 and not tested during this step.
- Both flags F. The subdomain is divided into two subdomains, which are put back on the queue.

Once at least one value is labeled with a 1 (tentatively irreplaceable), values in that domain are only tested if they are labeled with 0 or 1. However, if a formerly

Table 13. Efficiency of CNR Algorithms for Problems with Relop Constraints Using Inference and Ordering Strategies[a]

	graph density		
	0.2	0.3	0.4
CNR(Q)	128.7	452.1	293.0
CNR-queorder	115.1	237.0	307.0
CNR-inference	104.8	396.3	263.4
CNR-infer+queord	90.2	179.0	257.7

[a]Means for 50 problems. Times in sec.

irreplaceable value is found to be replaceable, and no other value has a label 1, then values labeled -1 are also tested. Moreover, if a situation is found where a value labeled with -1 is found between two values labeled with 1, then values labeled -1 are also tested. The same rules also apply to the values considered as replacements for the value being tested during the all-solutions search.

The soundness of these procedures can be deduced from Propositions 13-15. Consider, for example, the case where a replaceable value v is found with flags F and T on the 'left' (below smallest subdomain value) and 'right', respectively. In this case we know that values to the 'left' of v are all potentially replaceable, since the values on the 'right' may not all be, and Proposition 14 tells us that all values on one side are replaceable. Now, if a value below the largest subdomain value is later found to be replaceable, then it becomes possible that values to the 'right' of v are all replaceable as well. But then this continues to hold for values below v, so the status of those values remains unchanged. Note that in ambiguous cases such as a replaceable value between two T flags, the subdomain is simply divided and no attempt is made to infer replaceability. Note also that when a potentially irreplaceable value is found, labeling untested values does not preclude their being tested later in the procedure. And, finally, note that at the beginning of each step the end values of the current domain are tested to determine the initial flags.

A test of these procedures was done using small relop problems, which allowed the basic algorithm as well as algorithms employing more advanced techniques to be run in a reasonable time. These problems had 30 variables and domain size 8; constraints were either $\geq$ or $\neq$ in equal proportion. Several graph densities were tested; results for the hardest problem sets are shown in Table 13.

As a test of the soundness of the implementation, problems were retested and inconsistency and gap tests were carried out at the end of preprocessing. Occasional cases of irreplaceable sets with a gap were found with problems of the lowest density. These were hand-tested and shown to be complete, i.e. all other values were found to be replaceable given the state of the current adjacent domains.

These results show that, just as with the distance problems, improvements can be effected either by a queue ordering heuristic or by methods that deduce replaceability, although in this case the differences are not nearly as marked. Although the pattern of differences in efficiency varies as a function of graph density, in all cases the combination of queue ordering and inference gives the best results; improving times by more than a factor of 2 for the hardest problems.

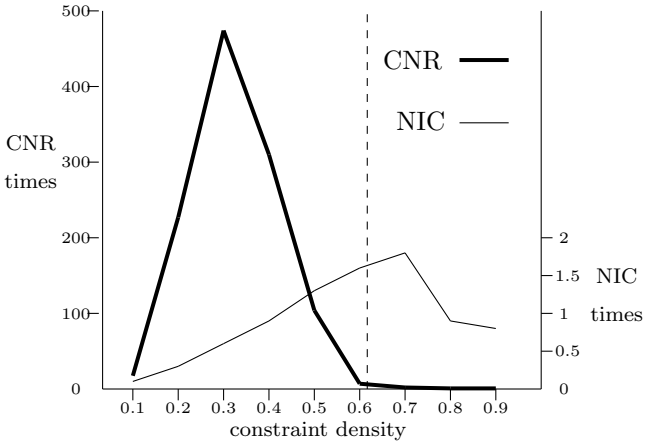


Figure 11. Runtime curves for CNRQ and NIC with relop problems of varying constraint density. Times are in seconds. Each point is the mean for 50 problems. Vertical dashed line indicates approximate transition (50:50) point from satisfiable to unsatisfiable problems.

11. Patterns of Difficulty Across the Problem Space

An interesting feature of replaceability algorithms that use depth-first search (which may also hold for other replaceability algorithms) is that there is a pattern of effort as a function of problem parameters that is remniscent of the well-known critical complexity effect for CSP search. An example is shown in Figure 11, where a curve for CNR runtime appears along with a corresponding curve for NIC. In this experiment, constraint density is the parameter that is varied. [2] As with other types of local consistency algorithms (Gent, MacIntyre, Prosser, Shaw, & Walsh, 1997), the curve for NIC reaches its highest point near the phase transition. In contrast, the peak of the CNR curve is well to the left of this point in the 'easy' region with respect to search effort. (Mean search effort following AC in terms of nodes was 106 at 0.5 density, 930 at 0.6, and 518 at 0.7, using min domain/forward-degree.)

Similar results were found with distance problems. With 'geovarsat' problems the results were not clear-cut, but in this case varying constraint density also changes patterns of connectivity (i.e. with more constraints the constraint graph becomes less clumpy). Hence, it isn't clear whether complexity effects like those found with other problem types should be found here as well.

To get some insight into the basis for this effect, tests were done on the same problems where various operations and other measures were counted, in particular number of neighbourhood solutions and queue additions. (Value removals were already counted.) While value deletions and queue additions did not show any clear relation to runtimes, there was a strong correlation with the number of neighbourhood solutions, as shown in Table 14.

Why are there so many neighbourhood solutions? With increasing density, neighbourhood size increases, and therefore the number of possible tuples grows as d^k, where k is the number of neighbours. But at the same time the number of constraints among neighbours will also increase. Given a particular variable with neighbourhood of size k, we can calculate the likelihood that an additional constraint will either produce a new neighbour or increase the number of constraints within the subgraph composed of the variable and its neighbours. The former is $\frac{1}{n} * \frac{(n-k-1)}{n} = \frac{(n-k-1)}{n^2}$, the latter is

[2] In the graph, density is in terms of the edges added to a spanning tree. Thus all graphs are connected. This means that 0 on the abscissa of Figure 11 represents a spanning tree, while 1 is as usual a complete graph.

Table 14. CNR Statistics for Relop Problems[a]

density	neigh-sols	removals	Q-addits[b]	prove unsol
0.1	297,025	193	99	–
0.2	3,889,129	101	67	–
0.3	9,564,750	82	57	–
0.4	6,706,596	95	58	–
0.5	1,809,104	122	72	1
0.6	72,054	152	70	17
0.7	2,236	79	20	44
0.8	5	14	1	50
0.9	–	–	–	50

[a]Same relop series as in Figure 11.

[b]Number of additions to original list of variables during replaceability testing.

$(\frac{k}{n})^2$. For problems with 30 variables, and density = 0.1 (in terms of the proportion of edges added to a spanning tree), the average degree is 5. This, in turn, gives a probability of about 0.027 for adding a new neighbour and 0.028 for adding a constraint to the neighbourhood subgraph. For density = 0.3, the average degree is about 10, and the two probabilities are approximately 0.021 and 0.111, respectively. So we can see a shift in relatively likelihood between these two densities. By setting the two expressions equal to each other, we can derive a value of k equal to 5 for the crossover point for problems of this size.

It might be argued that since there is no phase transition in this case, this cannot be considered a complexity peak. Although we can't yet be certain that there is a genuine asympototic complexity effect, for some classes of problems this may well be the case. The implication is that complexity peaks are a more general phenomenon than phase transitions. In fact, garden-path based complexity (backtrack search algorithms), support based complexity (consistency algorithms), and neighbourhood solution based complexity (CNR) share a basic property. All are related to a situation where the number of possibilities increases at the same time that the likelihood that a possibility is a viable member of the desired set is decreasing.

The present difficulty peak is obviously related to one that might be found for all-solutions search (depending on the problem model and manner of search). A significant difference is that, as shown in the table, the result obtained in this case (number of deletions) does not itself increase.

Beyond its potential theoretical significance, knowledge of such a pattern of difficulty is of great practical importance in using CNR algorithms. Before becoming aware of this effect we had thought it was impossible to test certain structured problems beyond a very small number of variables, because on testing with increasing density we quickly came up against very long runtimes, and we assumed that the situation would only get worse as density increased. Now we know that the limits are not quite as severe provided one stays away from the peak complexity.

12. Conclusions and Future Work

The goal of the present paper was to systematize and explore higher-level properties that can be used either to simplify problems or to reformulate them in useful ways. The major theme is substitutability; we have shown that there is a hierarchy that can be constructed around this theme. Because it is a well-behaved hierarchy, properties can be identified with certain levels that hold for all levels below but do not hold at higher levels.

We also verified earlier claims (Bordeaux et al., 2008; Jeavons et al., 1994) that the property we call replaceability has special significance. By locating this property within a hierarchy of substitutability properties, we have also clarified the basis for its significance, in that it is the most general property within this hierarchy that can support local forms of substitutability, which in turn can be used to infer corresponding global properties. In this way, we have modified the more pessimistic assessment of (Bordeaux et al., 2008) concerning the usefulness of this property.

Our experimental results show that algorithms for replaceability can in fact find numerous values with this property in various problem classes making it a much more powerful form of simplification than the substitutability property. In this paper, we also introduced new algorithms for establishing neighbourhood replaceability that are much more efficient than algorithms based on other approaches. In addition, one algorithm, CNRQ, is decidely more efficient than other algorithms, although the difference in efficiency can vary greatly depending on the problem type.

Nonetheless, finding practical methods for establishing replaceability remains a challenging problem for many problem classes. However, for certain types of constraint, we have been able to develop methods that avoid the (repeated) full search otherwise required. These methods can sometimes result in orders-of-magnitude improvements, thereby expanding the scope of replaceability algorithms, although to be sure it is still quite restricted in comparison with consistency techniques. [3]

As a sidelight, we have discovered a new critical complexity effect associated with neighbourhood replaceability algorithms and analyzed some of its features. This is of both theoretical and practical interest.

Finally, we note that the present experiments have brought out the usefulness of neighbourhood inverse consistency (NIC) for preprocessing very hard problems. This suggests that not enough attention has been paid to this strategy in the past.

All this aside, we think that the real opportunities for using 'higher-level' properties such as replaceability remain to be discovered. For example, in cases where an effort is made to compile CSPs into more compact forms such as MDDs (Hadzic, Hausen, & O'Sullivan, 2008), synthesis trees (Weigel & Faltings, 1999), or composite CSPs (Sabin & Freuder, 1996), the effort required to establish replaceability or other related properties may be obviated to a degree. (In fact, it should be possible to compile properties such as replaceability, so that instead of discarding such values, they are kept on hand as potential substitutes.) Also, in dynamic situations where information from previous situations can be reused, it may be possible to amortize the cost of establishing high-level properties like replaceability, thus facilitating the process of finding new solutions to new problems. Finally, it may be possible to find approximations to full replaceability that are reasonably effective whilst being more efficient.

There are still many other topics that need to be considered. An important one follows from the fact that, above inconsistency, none of these substitutability properties is associated with a unique fixpoint. Moreover, above substitutability, the simplified problems are not even isomorphic. This means that in each case there are minimal sets associated with the property (e.g. minimal irreplaceable sets), as well as smallest minimal sets. At present, this doesn't appear to be a topic of major import since the number of replaceable values was similar for different orders of processing. But we cannot rule out the possibility that larger differences can occur with some problems. Unfortunately, according to (Jeavons et al., 1994) finding 'the smallest substitutable

[3] A reviewer suggested that it may be possible to devise a form of bounds replaceability; although the matter isn't clear as yet, this might be particularly useful when constraints have properties like the relop problems used in these experiments.

subset of a constraint is as difficult as solving the original problem' (p. 6). [4] Nonetheless, this topic may still be worth exploring.

Finally, there is much work to be done concerning the relations between simple substitutability properties of the sort we have considered here and other properties such as forbidden patterns, some of which also allow local to global reasoning (e.g. (Cooper et al., 2014)). It is already clear that neither replacebility nor the broken triangle property (BTP) dominates the other; in fact, there are cases where one of these can remove many values while the other is ineffective (e.g. RLFAPs for BTP and geometric problems for replaceability). Understanding these differences in more depth is a challenging problem for the future.

Acknowledgements

This publication has emanated from research supported in part by a grant from Science Foundation Ireland under Grant No. 05/IN/I886 and Grant No. SFI/12/RC/2289, which were co-funded under the European Regional Development Fund. For the purpose of Open Access, the author has applied a CC-BY public copyright licence to any Author Accepted Manuscript version arising from this submission. We thank Nic Wilson for his comments on the formal definitions, and we acknowledge the contributions of anonymous reviewers whose incisive criticisms did much to enhance the quality of this paper.

13. Data Availability Statement

Data from experiments is available at ucc.insight-centre.org/rjw/wallaceJETAIdata.zip

14. Disclosure statement

The authors have no competing interests to declare.

References

Bordeaux, L., Cadoli, M., & Mancini, T. (2008). A unifying framework for structural properties of CSPs: Definitions, complexity, tractability. *Journal of Artificial Intelligence Research*, *32*, 607–629.

Boussemart, F., Hemery, F., Lecoutre, C., & Sais, L. (2004). Boosting systematic search by weighting constraints. In *Sixteenth European conference on artificial intelligence-ECAI'04* (pp. 146–150). IOS.

Bowen, J., & Likitvivatanavong, C. (2004). Domain transmutation in constraint satisfaction problems. In *Nineteenth national conference on artificial intelligence – AAAI'04* (pp. 149–154).

Cooper, M. C. (1997). Fundamental properties of neighbourhood substitution in constraint satisfaction problems. *Artificial Intelligence*, *90*, 1–24.

[4] Note that in this passage Jeavons et al. use the term 'substitutable' in a different way than we do in this paper, to refer to value(s) that can be substituted for a value or values.

Cooper, M. C. (2020). Strengthening neighbourhood substitutability. In H. Simonis (Ed.), *Principles and practice of constraint programming - CP 2020. LNCS no. 12333* (pp. 126–142). Springer.

Cooper, M. C., El Mouelhi, A., Terrioux, C., & Zanuttini, B. (2014). On broken triangles. In B. O'Sullivan (Ed.), *Principles and practice of constraint programming - CP 2014. LNCS no. 8656* (pp. 9–24). Springer.

Debruyne, R., & Bessière, C. (1997). Some practicable filtering techniques for the constraint satisfaction problem. In *Fifteenth international joint conference on artifcial intelligence – IJCAI'97. vol. 1* (pp. 412–417). Morgan Kaufmann.

Fredkin, E. (1960). Trie memory. *Communications of the ACM*, *3*, 490–499.

Freuder, E. C. (1991). Eliminating interchangeable values in constraint satisfaction problems. In *Ninth national conference on artificial intellgence – AAAI'91* (pp. 227–233).

Freuder, E. C. (2011). Dispensable instantiations in constraint satisfaction problems. In *Tenth international workshop on constraint modelling and reformulation – ModRef 2011.*

Freuder, E. C., & Elfe, C. D. (1996). Neighborhood inverse consistency preprocessing. In *Thirteenth national conference on artificial intelligence – AAAI'96. vol. 1* (pp. 202–208). AAAI/MIT.

Freuder, E. C., & Wallace, R. J. (2017). Replaceability and the substitutability hierarchy for constraint satisfaction problems. In C. L. C. Benzmueller & M. Theobald (Eds.), *Third global conference on artificial intelligence - GCAI 2017* (pp. 51–63). Easychair Publications.

Gent, I. P., MacIntyre, E., Prosser, P., Shaw, P., & Walsh, T. (1997). The constrainedness of arc consistency. In G. Smolka (Ed.), *Principles and practice of constraint programming-CP'97. LNCS no. 1330* (pp. 327–340). Springer.

Gent, I. P., Petrie, K. E., & Puget, J.-F. (2006). Symmetry in constraint programming. In F. Rossi, P. van Beek, & T. Walsh (Eds.), *Handbook of constraint programming* (pp. 329–376). Elsevier.

Gomes, C., & Walsh, T. (2006). Randomness and structure. In F. Rossi, P. van Beek, & T. Walsh (Eds.), *Handbook of constraint programming* (pp. 639–664). Amsterdam: Elsevier.

Hadzic, T., Hausen, E. R., & O'Sullivan, B. (2008). On automata, MDDs and BDDs in constraint satisfaction. In *ECAI 2008 workshop on inference methods based on graphical structure of knowledge.*

Jeavons, P., Cohen, D., & Cooper, M. C. (1994). A substitution operation for constraints. In A. Borning (Ed.), *Principles and practice of constraint programming - PPCP'94. LNCS no. 874* (pp. 161–177). Springer.

Johnson, D. S., Aragon, C. R., McGeoch, L. A., & Shevron, C. (1991). Optimization by simulated annealing: An experimental evaluation. part ii. graph coloring and number partitioning. *Operations Research*, *39*, 378–406.

Karakashian, S., Woodward, R., Choueiry, B. Y., Prestwich, S. D., & Freuder, E. C. (2010). A partial taxonomy of substitutability & interchangeability. In *Tenth international workshop on symmetry in constraint satisfaction problems - SymCon 2010.*

Likitvivatanavong, C., & Yap, R. H. C. (2013). Eliminating redundancy in CSPs through merging and subsumption of domain values. *ACM SIGAPP Applied Computing Review*, *13*, 20–29.

Mackworth, A. (1977). Consistency in networks of relations. *Artificial Intelligence*, *8*(1), 99–118.

Mackworth, A., & Freuder, E. C. (1985). The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, *25*, 65–74.

Sabin, D., & Freuder, E. C. (1994). Contradicting conventional wisdom in constraint satisfaction. In A. Borning (Ed.), *Principles and practice of constraint programming - PPCP'94. LNCS no. 874* (pp. 10–20). Springer.

Sabin, D., & Freuder, E. C. (1996). Configuration as composite constraint satisfaction. In *Configuration. papers from the 1996 AAAI fall symposium. Tech. Rep. FS-96-03* (pp. 28–36). AAAI Press.

Taillard, E. (1993). Benchmarks for basic scheduling problems. *European Journal of Opera-*

tional Research, *64*, 278–285.
Tsang, E. (1993). *Foundations of constraint satisfaction.* Academic Press.
Wallace, R. J. (2015). SAC and neighbourhood SAC. *AI Communications*, *28*, 345-364.
Wallace, R. J. (2018). Replaceability for constraint satisfaction problems: Algorithms, inferences, and complexity patterns. In A. S. D. Lee & T. Walsh (Eds.), *Fourth global conference on artificial intelligence - GCAI 2018* (pp. 243–255). Easychair Publications.
Weigel, R., & Faltings, B. (1999). Compiling constraint satisfaction problems. *Artificial Intelligence*, *115*, 257–287.

Appendix A. Proof of Proposition 13

In order to prove this proposition, we consider the situation for a single variable, and we consider the set of possible environments, i.e. neighbourhoods of that variable. With respect to greater-than or less-than constraints, we find that we can reduce these to a single equivalent constraint, which of course greatly reduces the number of cases. Then, we consider individual cases. In addition, by considering all possible patterns of domains, we can obviate the need to consider constraints between neighbours. While the simplest cases can be proven directly, in the end we must rely on testing by running the CNR program on more complex combinations of greater-than, less-than and inequality constraints. This entails testing all patterns of subranges of the original domains for the critical variable and its neighbours. (We do not have to consider other subsets because the propositions show that if we begin with domains that are complete subranges, this will always be the result when we reduce a domain to a set of irreplaceable values.) Because of combinatorial explosion, we are restricted to relatively small domains and numbers of inequality constraints; however, this is sufficient to cover some of the cases used in the experiments.

Also, since a $\geq$ ($\leq$) constraint is equivalent in its effects to a $>$ ($<$) constraint whose domain values are each shifted up (down) by 1, without loss of generality we only consider constraints of the form $>$ and $<$.

We begin by establishing the needed equivalences for $>$ and $<$ constraints:

Lemma 1. We consider binary CSPs with constraints based on relational operators other than equality, whose original domains are complete integer sequences and where there are no inequality constraints associated with singleton domains. And we consider a given variable X_i whose values are currently being tested for replaceability. Given a particular set of assignments to variables adjacent to X_i, call the set of possible assignments to X_i a viable assignment set. Then for the set $C_{ij_>}$ of $>$ constraints, a single constraint of the form $>$ whose adjacent variable includes values from the least smallest value among the original domains to the least largest value will allow the same viable assignment sets as the original set of constraints. Similarly, for the set $C_{ij_<}$ of $<$ constraints, an equivalent $<$ constraint is one whose adjacent domain is a range of values from the largest smallest to the largest largest value of the original domains.

Proof. Since the set of viable assignments associated with a set of $>$ constraints is limited by the smallest value assigned to an adjacent variable, then if a domain has no values greater than k_{minmax}, then any values in other adjacent domains greater than k_{minmax} will have no effect on the viable assignment set. Hence, for purposes of examining the possible viable assignment sets, they can be discarded. Similarly, the lowest value in any adjacent domain k_{minmin} will be associated with the smallest

possible set of viable assignments for X_i regardless of the other adjacent assignments. Hence the equivalent $>$ constaint must include the range of values between k_{minmin} and k_{minmax}. A similar argument holds for $<$ constraints. □

Lemma 2. A constraint of the form $>$ ($<$) that is equivalent to a set of $>$ ($<$) constraints in the sense of Lemma 1, will allow the same replacement sets for viable assignments of X_i as the tuples based on the original set of adjacent variables.

Proof. This follows from the fact that for any viable assignment a replacement set must be a subset of the remaining viable values. Since this form of constraint equivalence merely removes duplicates of the viable assignment sets, the set of replacement sets must also be the same. □.

Proposition 13. We consider binary CSPs based on relational operators other than equality constraints, whose original domains are complete integer sequences and where there are no inequality constraints associated with singleton domains either initially or after consistency testing. In addition, we assume that values in a domain are tested in lexical order; without loss of generality we assume an ascending order. We also assume that domain sizes do not exceed eight values, and the number of inequality constraints is four or less. Under these conditions, the set of irreplaceable values produced by a CNR algorithm is always a complete subrange.

Proof. We first consider the effects of adjacent constraints C_{ij} on a single variable, X_i, and we proceed by cases. Lemma 2 allows us to consider sets of $>$ plus $\geq$ constraints or $<$ plus $\leq$ constraints as a single constraint of these types.
Cases 1 and 2: With a single $>$ constraint, if there are consistent values, these will form a range in which all values of X_j supported by a larger value v_l in D_{x_i} will also be supported by values smaller than v_l. As a result the set of irreplaceable values becomes a singleton whose only member is the largest value smaller than the smallest value in D_{x_j} By the same reasoning, with a single $<$ constraint, the set of irreplaceable values will be the largest value in the range of consistent values.
Case 3: Both $>$ and $<$ constraints. We associate $>$ with X_{j_1} and $<$ with X_{j_2}. The consistent subrange of D_{x_i} is that between the smallest value greater than the smallest value of X_{j_2} and the largest value less than the largest value of X_{j_1}. Since each value is consistent with a value in X_{j_1} that is smaller than any supported by larger values and at the same time is consistent with a value in X_{j_2} that is larger than any supported by a smaller value, each value in the consistent subrange is irreplaceable.
Case 4a: A single inequality constraint; D_{x_j} is non-singleton. In this case, values can be only replaced if some values are not equal to any of the values in D_{x_j}, and in this case the resulting domain is a singleton. Otherwise, the domain of X_i is unchanged.
Case 4b: A single inequality constraint; D_{x_j} is singleton. In this case, any value not equal to the single value of X_j can replace any other value, so the set of irreplaceable values is a singleton.
Cases 6 and 7: A $>$ ($<$) constraint together with a $\neq$ constraint, the latter associated with a non-singleton domain.
The remaining cases, which involve varying numbers of $\neq$ constraints and differing ranges of values in the domain of X_i and its neighbours, were handled with a computer program designed to go through all possible cases. That is, all possible subranges of the set $\{1, 2, \ldots 8\}$ in all possible combinations were tested, where a test means that variable X_i being tested for replaceability has some subrange and is being tested against all possible combinations of subranges for the adjacent variables. This represents all possible configurations that a variable in the problem could be faced with when replaceability testing was carried out. It therefore covers all cases where reduc-

tions in adjacent domains induce further reductions in a given current domain. In these tests, the total number of $\neq$ constraints was varied from 2 to 4. In all cases, the resulting set of values in D_{x_i} after replaceability testing formed a complete subrange.[5]

Since the proposition holds for any variable and its neighbourhood within the specified limits, then under the specified conditions it will always hold during the consistency/replacement process. □

[5]The program is available as part of the supplemental material.