Dispensable Instantiations in Constraint Satisfaction Problems

Eugene C. Freuder, Diarmuid Grimes, and Richard J. Wallace

Insight Centre for Data Analytics
Western Gateway Building, University College Cork, Cork, Ireland
{e.freuder,d.grimes,r.wallace}@4c.ucc.ie

Abstract. ‘Simplifying’ problems, by removing values or combinations of values, has been a primary approach to combinatorial complexity in constraint satisfaction problems. This paper provides a unifying framework for much of this previous work, and in the process presents new opportunities. It extends a framework for CSP properties provided by Bordeaux, Cadoli, and Mancini. New forms of substitutability and subproblem extraction are introduced. An algorithm for one form of substitutability, neighbourhood replaceability, is presented along with preliminary experimental results.

1 Introduction

‘Simplifying’ problems, by using inference to remove values or combinations of values, has been a primary approach to combinatorial complexity in constraint satisfaction problems. The removal of inconsistencies, which will not eliminate any solutions, has been most thoroughly studied. However, often a single solution suffices, and methods such as interchangeability have been introduced, which remove some, but not all, solutions. This paper provides a unifying framework for much of this previous work, and in the process presents new opportunities. It extends the framework for CSP properties provided by Bordeaux, Cadoli, and Mancini in [1].

This simplification or “filtering” has often been viewed as part of the solution process, but it can also be viewed as a process of successive refinement or reformulation. Achieving arc consistency before search, for example, can be viewed as preprocessing the problem to produce a more efficient model for the subsequent search, while an algorithm like MAC, which interleaves arc consistency processing with search, can be viewed as employing a form of “dynamic reformulation”.

The fundamental property introduced in this paper is “dispensability”. Its simplest form is value dispensability. A value for a variable is dispensable for a problem P if removing the value will not remove all solutions of P; otherwise it is

Proceedings of the 21st RCRA workshop on *Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion* (RCRA 2014).
Vienna, Austria, July 17–18, 2014.

indispensable. While we will focus on solvable problems, these definitions apply to unsolvable problems viewed as an ‘extreme case’: for unsolvable problems all values or instantiations are dispensable because removing them cannot remove all solutions; there are not any solutions to remove.

At first sight it might seem like dispensability is a vacuous concept: in many problems every value will be dispensable, removing any one of them individually will not remove all solutions. At best we only need retain a single value for every variable to ensure a single solution. However if we knew such a set of single values we would have solved the problem. In practice, it can be very useful to remove values that we can identify as dispensable in some tractable manner, either in preprocessing before search, or dynamically during search. This is, for example, what the basic process of ensuring arc consistency does. (In that case, removing an inconsistent value will not remove all solutions for the simple reason that the value will not participate in any solution.)

The very general concept of dispensability supports a context for viewing many forms of constraint processing. This provides opportunities for identifying new concepts to ‘fill in holes’ or applying established methods by analogy. We will see several examples. Section 2 defines various forms of dispensability, extending Bordeaux et al., in particular beyond values to instantiations and subproblems. Section 3 finds new applications of methods for tractably computing local forms of dispensability. Section 4 presents an algorithm for one form of substitutability and some experiments on discarding dispensable values based on this form of substitutability. Section 5 gives conclusions.

2 Dispensability

A constraint satisfaction problem (CSP) involves choosing values for a set of variables, V, which satisfy a set of constraints, C, which specifies permissible combinations of values for subsets of the variables. A choice of values for a subset, S, of the variables is an instantiation of S; an instantiation of all the variables, V, which satisfies all the constraints is a solution.

Definition 1. An instantiation is *dispensable* if removing it will not remove all solutions to the problem. A set of instantiations is dispensable if removing them all will still not remove all solutions to the problem.

To simplify matters, we will assume that constraints are represented explicitly as sets of instantiations, representing allowed combinations of values, and removing an instantiation of S can be accomplished by deleting it from the constraint on S. If we view the domain of values for a variable as a unary constraint, a dispensable value may be viewed as a dispensable instantiation, where S contains a single variable. Note that as we remove dispensable instantiations, we effectively generate a sequence of problems, each of which has a solution set that is a subset of the previous problem. Dispensability is defined relative to a specific problem; an instantiation that is dispensable in the initial problem may become indispensable at some subsequent point in the sequence. Note also that if a problem is inconsistent, any instantiation is dispensable, because there are no solutions to remove.

Dispensable instantiations are of particular interest because their removal can reduce the search effort required to find a solution. (However, the savings has to be balanced against the effort required to identify the dispensable instantiations, and indeed removal may perversely make matters worse [8]). Work has also been done on removing entire "redundant" constraints [2], but this is equivalent to adding all the missing instantiations to a constraint, and will be beyond the scope of this paper.

2.1 Substitutability

Many specific forms of dispensability can be viewed as variations on the theme of substitutability [3]. A value u is *substitutable* for a value v if for any solution in which v appears, we can substitute u and still have a solution. In this case, v is clearly dispensable, and we will say that v is *substitutable.*

Interchangeability [3] is a stronger form of substitutability; two values are interchangeable if each is substitutable for the other. Given two interchangeable values, we can dispense with either one. Interchangeability will remove fewer values, but the advantage is that we can more easily recover the discarded values than with substitutability. Interchangeability and substitutability have primarily been studied for values but have been extended to instantiations in the context of conditional interchangeability and substitutability [16]. Jeavons, Cohen and Cooper applied a form of substitutability to instantiations (labelings) earlier in [10].

[1] defines a value v as *removable* if for any solution involving v, there is some other value that can be substituted for v and still have a solution. Clearly this definition can be extended to instantiations. Removability can be regarded as a weaker form of substitutability, where the substitution does not always have to be the same. [1] regards removability as "fundamental" and seems to view it as equivalent to disposability: "Another central role is played by the removability property, that characterises precisely the case when a value can be safely removed from the domain of a variable, while preserving satisfiability". However, clearly a value can be dispensable without being removable. Consider, for example, a problem with two solutions, which have no values in common. Any value is dispensable, none are removable. This rather makes "removable" something of a misnomer; "replaceable" might be preferable. Moreover we can define a further weakening of substitutability that lies between removability and dispensability:

Definition 2. An instantiation v is *minimally substitutable* iff if v is in any solutions there is some other instantiation we can substitute in at least one of them and still have a solution.

We now have a hierarchy, as we proceed to the right, we can remove more instantiations, but it becomes harder to recover the lost solutions:

interchangeability $\rightarrow$ substitutability $\rightarrow$ replaceability $\rightarrow$ minimal substitutability $\rightarrow$ dispensability.

The most studied form of dispensability is *inconsistency* [12]. An instantiation is inconsistent if it does not appear in any solution. For example, path inconsistency removes instantiations from binary constraints. Inconsistency can actually be viewed as an extreme form of substitutability. "Any solution in which v occurs" just refers in this case to the empty set. If there are no solutions, any value is substitutable for any other. Inconsistency implies interchangeability in the sense that two inconsistent values for a variable are interchangeable. However, a value can be inconsistent without being interchangeable if it is the only inconsistent value for that variable.

Dispensability itself can also be viewed as an extreme form of substitutability at the other end of the spectrum. Minimal substitutability of course implies dispensability. We will now characterize dispensability as a specific form of substitutability.

Partial interchangeability for two values of variable X with respect to a subset S of variables is defined in [3] as any solution involving one implies a solution involving the other with possibly different values for S.

Definition 3. An instantiation v for a set of variables T is *minimally partially substitutable with respect to a set of variables S* iff if v is in any solution, there is at least one solution involving some other instantiation for T, and possibly other values for variables in S.

Proposition 1. Given an instantiation v for a set of variables T, and the set R of all variables other than T, v is minimally partially substitutable with respect to R iff v is dispensable.

While we have now shown that everything on the spectrum from inconsistency to dispensability can be viewed as a form of substitutability, the two extremes represent almost perverse interpretations, where an actual substitution is essentially irrelevant. The concept of dispensability itself seems to better capture the essential spirit of our endeavor here – to simplify by removal – and thus may be better able to motivate new concepts like minimal substitutability.

2.2 Sufficiency

We now define a concept of sufficiency, which will help us identify sets of indispensable instantiations.

Definition 4. An instantiation is *sufficient* iff if the problem has a solution it has a solution containing that instantiation.

Proposition 2. If an instantiation for a subset of variables, S, is sufficient, the set of all other instantiations for S is dispensable.

We can establish sufficiency by examining the "inverse" of properties discussed in the previous subsection. A consistent instantiation appears in at least one solution. Fixable and implied (equivalent to indispensable) are defined for values in [1]; we extend the definitions here to instantiations.

Definition 5. An instantiation is *indispensable* iff it appears in all solutions. An instantiation is *fixable* iff it is substitutable in all solutions.

Proposition 3. If an instantiation for a subset of variables, S, is consistent, indispensable, or fixable it is sufficient.

2.3 Subdomain Subproblems

Subproblems present another opportunity for removing a whole set of dispensable instantiations at once.

Definition 6. A problem Q is a *subdomain subproblem* of P iff P and Q have the same variables, with the domains of values in Q subsets of the domains of values in P, and the constraints of Q restricted to instantiations involving the domain values of Q.

Definition 7. A *subdomain subproblem of P is dispensable with respect to P* iff the Cartesian product of its domains, viewed as a set of instantiations for P, is dispensable with respect to P.

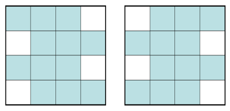
[7] provide a general method for extracting subdomain subproblems by transforming a problem into a disjunction of subproblems, none of which contain the dispensable instantiations. They explored two specific types of dispensable subproblems: *inconsistent subproblems* [7], which do not contain any solutions, and *consistent subproblems* [6], where the domains are restricted to all but one value, v, for a variable X, and all the values consistent with v for every other variable. The latter are dispensable because it can be shown that they do not contain all the solutions.

We have emphasized here that dispensability, including substitutability, is not just for values, but can be generalized to instantiations. However, while dispensable instantiations could be removed from the constraint on the entire set of variables involved in the instantiations, this may require replacing an implicit 'anything allowed' constraint with an explicit constraint that is too big to store and too costly to check. Viewing the set of instantiations as a subdomain subproblem allows us to use extraction to remove the instantiations without creating such constraints. This can be particularly relevant to restart algorithms, allowing lessons learned about dispensability in one solving attempt to be implemented in the next [13]. Extraction could also prove especially practical in a distributed environment in which we can repeatedly extract while distributing the resulting disjunctions of subproblems to different processors.

We consider now two specific new ways in which the dispensability framework suggests additional uses for subdomain subproblem dispensability.

Iterative Broadening. [9] introduced an iterative broadening approach to constraint satisfaction, where we start by only considering two values from the domain of each variable, if we fail we retry allowing three values, and repeat until successful (or failure on the full problem). Iterative broadening can, in particular, allow us to search using *only* the k best values for each variable, according to a heuristic criterion, before considering *any* inferior values. Ginsberg and Harvey demonstrated that iterative broadening somewhat counter intuitively can be remarkably successful even though it can clearly involve repeatedly redoing a great deal of work. Extracting subdomain subproblems as we iterate allows us to utilize iterative broadening while avoiding that redundant search effort. In moving from failure on the problem with domain size restricted to k to an attempt on the problem with domain size $k+1$ we can extract the subdomain subproblem corresponding to the former from the latter. (We are assuming a fixed, heuristic value ordering.) If we assume all domains have the same size, we can reduce the search space at each iteration from $(k+1)^n$ to $((k+1)^n - k^n)$.

Solution-preserving Mapping. Consider the classic 4-Queens problem. The shaded areas in the two boards below represent two subdomain subproblems. If we map one onto the other by flipping around the vertical axis of symmetry, any solution to the full problem in one will be a solution in the other. Thus we can extract those instantiations that are not common to both. We can remove $3^4 - 2^4$ or 65 of the 256 possible choices. We keep the instantiations in the 'intersection' of the two subproblems in case the only solutions are in the intersection. (However, in this particular example, it is easy to see that there are no solutions in the intersection, and we could, in fact, remove all 3^4 or 81 instantiations of one of the subproblems.)



Proposition 4. If f is a solution-preserving mapping from one subdomain subproblem of P, SP1 onto another, SP2, then either subproblem can be extracted from P, if we add to the resulting disjunction of subproblems another disjunct consisting of the subdomain subproblem where the domain of each variable is the intersection of the domains of that variable in SP1 and SP2.

3 Tractability

Removing dispensable instantiations, either in preprocessing or dynamically during search, can reduce search effort. This has been amply demonstrated in practice. In general, however, determining dispensability can be as intractable as solving the problem can be. One way to address this is to consider restricted classes of problems, where the computation of forms of dispensability is tractable [1]. We focus here instead on situations in which dispensability with respect to a subproblem implies dispensability for the full problem, and thus the complexity is tractable in the sense of being limited by the size of the subproblems we consider.

3.1 Hierarchical Subproblems

Definition 8. The subproblem of P *induced* by a subset, S, of the variables of P, is comprised of S and the constraints that only involve variables in S.

Proposition 5. If an instantiation is inconsistent for an induced subproblem of P it is inconsistent for P.

K-consistency [4] and inverse k-consistency [5] provide induced subproblem hierarchies that successively identify more inconsistencies at successively higher cost. Both look at induced subproblems with k variables. K-consistency asks if instantiations for k-1 variables are consistent in the subproblem; inverse k-consistency asks if instantiations for 1 variable are consistent in the subproblem.

Inverse k-interchangeability is defined (for binary CSPs) in [3], where it is shown that it implies interchangeability for the full problem. Unfortunately it is termed k-interchangeability there. We provide here appropriate definitions for both k-interchangeability and inverse k-interchangeability.

Definition 9. A value v, for variable V, is *inverse k-interchangeable* with another u iff it is interchangeable with u in any subproblem induced by V and k-1 other variables.

Definition 10. An instantiation s, for a set S of k-1 variables, is *k-interchangeable* with another instantiation t iff it is interchangeable with t in any subproblem induced by S and 1 other variable.

Proposition 6. k-interchangeability implies interchangeability.

Proof. Given k-interchangeable instantiations i1 and i2 of a set X of k-1 variables. Consider any solution, s, of the entire problem that contains i1. Pick a value, v, in s, for any variable V not in X. This value v will be consistent with all the values in i1; thus i1 plus v will be a solution to the subproblem induced by X plus V. Since i1 and i2 are k-interchangeable, i2 plus v will also be a solution to this subproblem and thus i2 is consistent with v. Therefore i2 can replace i1 in s and we will still have a solution. Similarly i2 can be exchanged for i1 in any solution. □

Values v and u for variable V are *neighbourhood interchangeable* if they are consistent with the same values in every variable that shares a constraint with V. Both k-interchangeability and inverse k-interchangeability are equivalent to neighbourhood interchangeability when k=2. Neighbourhood interchangeabilty is defined for binary CSPs in [3], where it is shown that all neighbourhood interchangeable values can be identified in quadratic time, and demonstrated analytically that removing these values can achieve exponential savings, and is arbitrarily effective in the sense that whatever savings is specified, there is a CSP for which it saves that amount of effort.

3.2 Closure Subproblems

[1] claims that removability, what we call here replaceability, cannot be "detected efficiently, but incompletely, through local reasoning"; this, they say, justifies the

extensive use of properties like inconsistency and substitutability, which can. The paper raises “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, [1] uses a narrow definition of “local reasoning”. We show now that removability or replaceability can itself be computed locally in a manner analogous to the local computation of neighbourhood inverse consistency [5].

Definition 11. Let S be a subset of the variables of problem P, and PS be the subproblem induced by those variables. The *closure* of PS, CPS, 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 CPS the *frontier variables.*

Definition 12. An instantiation of variables S is *closure-replaceable* if it is replaceable with respect to the closure of the subproblem induced by S.

Closure-replaceability is related to the concept of local substitutability defined in [10]. The authors there define local substitutability in terms of “substitutable for a constraint”, which is not defined precisely; but, bearing in mind that the ‘substitutable’ instantiations in this paper are the ones to be eliminated, while in [10] they are the ones to be kept, roughly speaking the situation seems to be this: Closure-replaceability appears to be a generalization of local substitutability. Their “substitutablity” on the other hand appears to be a generalization of the present “replaceability”.

Proposition 7. 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. Given a problem P, and an instantiation v for variables S that is closure-replaceable, where CPS is the closure. Suppose sp is a solution of P that contains v; sp must also contain a solution, scps, to CPS that includes v. Since v is replaceable with respect to CPS, there is some other instantiation u of S that can be substituted for v in scps yielding another solution for CPS. When we substitute u for v in sp we obtain another solution for P, since the constraints involving S are all in CPS. □

Of course, in general the closure can be the entire problem, but 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 13. An instantiation of a single variable V is *neighbourhood replaceable* if it is replaceable with respect to the closure of the subproblem induced by V and its neighbouring variables.

Recently a new form of singleton arc consistency (SAC) was proposed and examined extensively in [15]. In this form of SAC, arc consistency is established following reduction of a domain of one variable to one value, but only with respect to the subgraph formed by that variables and its neighbours. In other words, here again S is a single variable, X_i, and each of its values is considered singly. If, for a given value a in the domain of X_i, a domain wipeout occurs, then this value can be removed. But, of course, being singleton neighbourhood inconsistent, it is also neighbourhood-replaceable. This gives us the following proposition.

Proposition 8. Neighbourhood replaceability implies neighbourhood singleton arc consistency.

4 Effectiveness

As with stronger forms of inconsistency processing, there is the question of how much substitutability processing is cost-effective in practice. This devolves into more specific questions, such as:

– How many dispensable values are there corresponding to a given level of substitutability, in a problem of a given type?
– How much effort is required to find and remove them?
– What is the effect of such removal on subsequent search effort?

These are large questions. Much of the history of constraint programming has concerned efforts to address these questions for different forms of inconsistency, including the development of many efficient or specialized methods to remove inconsistent values. A variety of interchangeability/ substitutability methods have also been studied analytically and experimentally. Here we will present some initial analysis with regard to neighbourhood replaceability, focusing on the first of the three questions above.

A basic algorithm for establishing neighbourhood replaceability is given in Figure 1. It is called the "consistent neighbourhood replaceability" algorithm (CNR) because it removes all values that are arc consistent *and* replaceable (in addition to removing arc-inconsistent values). In its full form it thereby incorporates neighbourhood SAC. Here, for analytical purposes we also wish to examine the effects of removing consistent replaceable values. In the current implementation, the replaceable procedure uses a MAC-style search to find all solutions for the subproblem; for each solution it seeks a value from the current domain that can replace value v.

We argue analytically, and verify experimentally, that CNR is unlikely to be useful anywhere in the standard random problem search space. At the same time, we need to bear in mind the limitations of the standard random problem model. We present evidence suggesting that CNR is more effective in removing dispensable values for more inhomogeneous problems, which arguably are more likely to be encountered in the real world.

```
Set Q to list of all variables
While not empty Q
    Remove V from Q; set S to neighbours of V
    For each value v in domain of V
        Set domain of V to {v}
        If arc-inconsistent({v}+S) or replaceable(v,S)
            Remove v from domain
            Put neighbours of V in Q if not there already
```

Fig. 1. Algorithm for neighbourhood replaceability.

The analytic argument is straightforward. Given a consistent neighbourhood solution s_n with value v_f belonging to variable V_f (f is for "focal"), we seek the probability of a value $v_{f'} \neq v_f$ such that $v_{f'}$ is consistent with the neighbouring assignments of s_n. This probability is equal to the number of neighbouring assignments, C, raised to the power $(1 - p2)$, where $p2$ is as usual the expected tightness of a constraint, i.e. $C^{(1-p2)}$. (Here, we assume that $p2$ 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 $(d - 1) \times C^{(1-p2)}$.

For example, suppose $d - 1 = 10$ and $p2 = 0.369$, so $(1 - p2) = 0.631$, 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	$C^{(1-p2)}$	exp. # replacem's
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 was verified in an experiment using homogeneous random problems of 50 variables with domain size 15 and 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. (This made these problems more commensurate with other classes of problems that are defined below.) 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. In these and all later samples, all problems had solutions. Since the purpose was to determine the number of neighbourhood replaceable values that were *not* also arc- or neighbourhood-inconsistent, only values that also met these criteria for consistency were counted.

Table 1. Values Removed from Problems of Varying Density

density	.10	.15	.17
mean values removed	43.7	3.9	0.6
# affected problems	25	14	2
mean degree of affected vars	1.86	2.12	2.00

1st line is mean per problem

Thus, for problems with sparse constraint graphs there are many such values, but this is because there are many variables with degree ≤ 2. (Occasionally, a single value was deleted from a variable of degree 3.) 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 search 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 parameter values. In particular, for each value in each domain the likelihood of support across each adjoining constraint is identical. But this is not usually realistic. And, in fact, problems can be generated that are still based on random selections but where the probability of support varies, for a given constraint or for all constraints that a value participates in. In the present work this was done by a two-tier 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. Note that in this scheme selection of a probability of support is done independently for each value and for each constraint. Otherwise, problems had the same parameters as the 0.17 density problems in the previous experiment.

In addition, problems can be generated in which the constraint graph is clumpy (so-called "geometric problems" [11]). These problems are generated by selecting points at random within the unit square and then joining those whose Euclidean distance is less than some criterion, called the "distance". In this case, the distance was 0.275. In addition, if there were $\geq$ one connected component, separate components were connected via the closest variables to make a single connected graph. Problems were generated with the same number of variables, domain size and expected tightness as the random problems (see table) and were filtered to select those with a number of constraints close to that for random problems with density = 0.17, specifically, the expected number $\pm$ 5.

Results are shown in the next table. Clearly irregularity of both kinds is 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.)

Table 2. Values Removed from Random and Geometric Problems with Values of Varying Tightness

	random		geometric	
problems	2-6/8-4	2-7/8,9-15	geo	geo:2-6/8-4
mean values removed	3.7	5.7	6.8	27.6
# problems with removals	15	21	17	25
mean degree of affected vars	3.28	3.56	5.74	6.59

Headers show support and proportion with this support. Thus, "2-6/8-4" means support of 0.8 with prob=0.6 and support of 0.2 with prob=0.4

Although 28 out of 750 values is a small proportion, it is interesting 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. In addition, the cost for finding them is not large; for these problems CNR never took more than a few seconds to complete its task. Moreover, with better knowledge of where replaceable values are located, it may be possible to reduce the effort appreciably by only checking where calculations indicate that the likelihood of finding such values is high enough to be worthwhile.

As one might expect, search efficiency is not greatly affected by deleting this small number of values. (And with MAC-3 and using the minimum domain/forward-degree heuristic, these problems were easy to solve.) Thus, in the following table only one column shows any difference at all, but interestingly this was due to one problem that was more difficult to solve.

Table 3. Search Efficiency on Random and Geometric Problems with and without Removing Consistent Replaceable Values

	random		geometric	
problems	2-6/8-4	2-7/8,9-15	geo	g:2-6/8-4
none	252.7	90.2	516.6	119.1
replaceab	252.7	90.4	497.6	120.8

Same problems as previous table. Mean search nodes per problem.

To follow up on this, we examined larger problems of the same types. Parameters for random problems were <80,15,0.077,.2-.6/.8-.4>, using the usual $< n, |d|, p_1, p_2 >$ notation, where the composite p_2 value has the same meaning as in the tables above. Geometric problems had the same number of variables, domain size, and composite tightness. However, it was not possible to generate problems with solutions having the same number of constraints as the random problems (316). Two sets of problems were generated, using distances of 0.17

and 0.18 and targets for number of constraints of 250 and 270, respectively (± 5). In this case, since most problems with solutions were very easy, 300 problems were generated from which problems with solutions were culled, and then the 25 problems that were hardest for the standard algorithm (MAC-3 with min domain/forward-degree) to solve.

Finally, we also looked at 120-variable geometric problems with varying support (geo-varsat problems). These problems had domain size 20, a distance of 0.17, a target of 540 (± 3) for number of constraints, which gives a graph density of 0.076, and a .2-.7/.8-.3 pattern of support. Three hundred problems were generated, of which 169 had solutions; results for this subset are reported here. Following the various types of preprocessing, search was done with MAC using the minimum domain/forward degree heuristic, with a cutoff of one million search nodes.

Table 4. Search Efficiency on Larger Problems with Various Forms of Consistency and Replaceability

problems	rand			geo:250			geo:270		
	nodes	time	rem	nodes	time	rem	nodes	time	rem
none	1725	11.1	–	560	0.9	–	66,184	148.1	–
replaceab	1721	15.7	9	163	6.7	123	25,900	72.9	96
AC	2621	19.1	72	232	0.3	57	52,084	126.4	61
NSAC	2540	18.3	88	90	1.7	370	94	2.0	500
CNR	2540	23.6	98	89	8.7	528	90	9.2	648
NIC	2540	23.2	87.5	90	4.6	374.8	92	5.2	505.6
NIC+CNR	2540	27.9	98.0	88	9.4	532.6	89	9.8	651.0

Problems as described in text above. Means per problem: search nodes, time (sec), values removed during preprocessing.

These results show that when sufficient numbers of values that are fully replaceable are discarded before search, this can have a significant effect on subsequent search effort. In some cases, the effect exceeds that of preprocessing with arc consistency. There is, of course, a price to be paid, as reflected in total run-time, but for more difficult problems, this preprocessing cost can lead to a much larger cost reduction during search.

In one case, there is a perverse effect of AC preprocessing, previously identified by [14], which also appears with other preprocessing algorithms. That this is due to the effect of preprocessing on search order heuristics was shown by employing a static degree heuristic to select the first variable and min domain/forward-degree thereafter. This reduced the mean number of search nodes to 1456 for the same problems after AC preprocessing and 1281 after preprocessing with NIC. (That this was not a generally superior search order was shown by the results for the other two problem sets; in both cases the size of the search tree increased by about 50% with this order.)

In evaluating search efficiency, we also looked at effects of preprocessing with neighbourhood inverse consistency (NIC) (cf. [5]) and with replaceability in addition to NIC. Here, an important question is whether CNR is able to delete values above and beyond those removed by NIC. In fact, one possibility is that NIC-preprocessing will result in more values that are fully replaceable.

Table 5. Search Efficiency and Values Removed on 120-variable Geo-Varsat Problems

	nodes	time	rem	>cut
AC	55,896	408	38	8
Replace	42,352	298	59	6
NSAC	30,602	154	189	5
CNR	30,277	214	269	5
NIC	10,211	97	310	1
NIC+CNR	8,788	122	383	1

Problems as described in text above. Entries as in Table 4, plus # exceeding cutoff. Means for nodes include cutoff values.

For the geometric problems, NIC reduced search effort dramatically. For these problems, when CNR was combined with NIC, the former algorithm removed even more values than it had without such preprocessing. In these cases, NIC removed over 4-500 values per problem, but adding neighbourhood replaceability reliably removed more values than when it was used alone (Table 5 versus 4). Somewhat surprisingly, when NIC was used the geometric problems became easy to solve; hence, adding replaceability in this case made little difference in search effort. Nonetheless, the fact that so many additional values could be removed is a promising result, which may improve search efficiency with still larger problems.

5 Conclusions

[1] present a powerful framework for viewing basic concepts in constraint processing. This paper extends and improves upon that framework, in the process identifying some intriguing opportunities for further study. Our experiments demonstrate that a certain kind of dispensable value can be located in a reasonably efficient manner in practice and that such values are not uncommon in certain classes of problems. Moreover, we have been able to demonstrate that removing such values can lead to significant improvements in search efficiency on some types of problems.

Acknowledgements. Some of this work was supported by the Science Foundation Ireland under Grant No. 05/IN/I886.

References

1. L. Bordeaux, M. Cadoli, and T. Mancini. A unifying framework for structural properties of csps: Definitions, complexity, tractability. *J. Artif. Intell. Res.*, 32:607–629, 2008.

2. A. Dechter and R. Dechter. Removing redundancies in constraint networks. In *Proc. NUMBER Nat. Conf. on Artif. Intell. – AAAI'87*, pages 105–109, 1987.
3. E. E. C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proc. 9th Nat. Conf. on Artif. Intell. – AAAI'91*, pages 227–233, 1991.
4. E. C. Freuder. Synthesizing constraint expressions. *Comm. ACM*, 21(11):958–966, 1978.
5. E. C. Freuder and C. D. Elfe. Neighborhood inverse consistency preprocessing. In *Proc. 13th Nat. Conf. of the Amer. Assoc. Artif. Intell. – AAAAI'96. Vol. 1*, pages 202–208. AAAI/MIT, 1996.
6. E. C. Freuder and P. D. Hubbe. Using inferred disjunctive constraints to decompose constraint satisfaction problems. In *13th Intl. Jt. Conf. on Artif. Intell. - IJCAI'93*, pages 254–261, 1993.
7. E. C. Freuder and P. D. Hubbe. Extracting constraint satisfaction subproblems. In *14th Intl. Jt. Conf. on Artif. Intell. - IJCAI'95*, pages 548–557, 1995.
8. E. C. Freuder, P. D. Hubbe, and D. Sabin. Inconsistency and redundancy do not imply irrelevance. In *AAAI 1994 Fall Symposium on Relevance*, AAAI Tech. Report FS-94-02, pages 74–78, 1994.
9. M. L. Ginsberg and W. D. Harvey. Iterative broadening. *Artif. Intell.*, 55(2-3):367–383, 1992.
10. P. Jeavons, D. Cohen, and M. C. Cooper. A substitution operation for constraints. In A. Borning, editor, *Principles and Pract. of Constraint Programming - PPCP'94*, number 874 in LNCS, pages 161–177. Springer, 1994.
11. D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Shevron. Optimization by simulated annealing: An experimental evaluation. part ii. graph coloring and number partitioning. *Opns. Res.*, 39:378–406, 1991.
12. A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
13. D. Mehta, B. O'Sullivan, L. Quesada, and N. Wilson. Search space extraction. In *Principles and Pract. of Constraint Programming - CP 2009*, LNCS 5732, pages 608–622. Springer, 2009.
14. D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A. Borning, editor, *Principles and Pract. of Constraint Programming - PPCP'94*, number 874 in LNCS, pages 10–20. Springer, 1994.
15. R. J. Wallace. Sac and neighbourhood sac. *AI Communications*, 27:to appear, 2014.
16. Y. Zhang and E. C. Freuder. Conditional interchangeability and substitutability. In *4th Intl. Workshop on Symmetry and Constraint Satisfaction Problems - SymCon'04*, pages 95–100, 2004.