Dispensable Instantiations

Eugene C. Freuder[1]

[1] Cork Constraint Computation Centre,
Department of Computer Science, University College Cork, Cork, Ireland
e.freuder@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 the framework for CSP properties provided by Bordeaux, Cadoli, and Mancini in [1]. New forms of substitutability and subproblem extraction are introduced.

Keywords: inconsistency, substitutability, interchangeability, reformulation, subproblems, subproblem extraction

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 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 [1], in particular beyond values to instantiations and subproblems. Section 3 finds new applications of methods for tractably computing local forms of dispensability.

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. 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 [2]). Work has also been done on removing entire "redundant" constraints [3], 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 [4]. 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 [4] 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 were extended to instantiations by Freuder and Zhang in the context of conditional interchangeability and substitutability [6]. Jeavons, Cohen and Cooper applied a form of substitutability to instantiations (labelings) earlier in [13].

[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; *onto-substitutable* might be preferable. Moreover we can define a further weakening of substitutability that lies between removability and dispensability:

Definition. 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->substitutability->onto-substitutability->minimal substitutability.

The most thoroughly studied form of dispensability is *inconsistency* [5]. 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 [4] as any solution involving one implies a solution involving the other with possibly different values for S.

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

Proposition. 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 2.1. 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. An instantiation is *indispensable* iff it appears in all solutions. An instantiation is *fixable* iff it is substitutable in all solutions.

Proposition. 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. 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. 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.

Freuder and Hubbe [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* [8], 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.

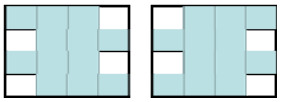
We have emphasized here that dispensability, including substitutability, is not just for values, but can be generalized to instantiations. However, while dispensable instantiations could be removed from the constraint on the entire set of variables involved in the instantiations, this may require replacing 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 [9]. 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. Ginsberg and Harvey [10] 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. (It turns out that in this case we need not have worried that could be a solution in the "intersection", but that is not true in general.)



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

K-consistency [11] and inverse k-consistency [12] 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 [4], 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. 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. 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. k-interchangeability implies interchangeability.

Proof. Given k-interchangeable instantiations i1 and i2 of a set X of k-1 variables. Consider a 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 *neighborhood 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 neighborhood interchangeability when k=2. Neighborhood interchangeabilty is defined for binary CSPs in [4], where it is shown that all neighborhood 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 onto-substitutability, *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 onto-substitutability can itself be computed locally in a manner analogous to the local computation of neighborhood inverse consistency [12].

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

Closure-onto-substitutability is related to the concept of local satisfiability defined in [13]. 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 [13] they are the ones to be kept, roughly speaking I believe the situation is this: Closure-onto-substitutability appears to be a generalization of local substitutability. Their 'substitutablity' on the other hand appears to be a generalization of my "onto substitutability".

Proposition. Closure-onto-substitutability implies onto-substitutability.

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-onto-substitutable, 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 onto-substitutable 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 and subproblems consisting of a single variable, i.e. we are looking for dispensable values. The closure will be the subproblem induced by that variable and all the neighboring 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 neighbors, then the complexity of the effort required to look for closure-onto-substitutable values is correspondingly bounded.

3 Conclusion

Bordeaux, Cadoli, and Mancini [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.

Acknowledgments. This material is based upon works supported by the Science Foundation Ireland under Grant No. 05/IN/I886.

References

1. Bordeaux, L., Cadoli, M., Mancini, T.: A Unifying Framework for Structural Properties of CSPs: Definitions, Complexity, Tractability. J. Artif. Intell. Res. (JAIR) 32, 607-629 (2008)
2. Freuder, E.C., Hubbe, P.D., Sabin, D.: Inconsistency and Redundancy Do Not Imply Irrelevance. AAAI 1994 Fall Symposium on Relevance, AAAI Tech. Report FS-94-02, 74-78 (1994)
3. Dechter, A., Dechter, R.: Removing Redundancies in Constraint Networks. AAAI 1987, 105-109 (1987)
4. Freuder, E.C.: Eliminating Interchangeable Values in Constraint Satisfaction Problems. AAAI 1991, 227-233 (1991)
5. Mackworth, A.K: Consistency in Networks of Relations. Artif. Intell. 8(1), 99-118 (1977)
6. Zhang, Y., Freuder, E.C.: Conditional Interchangeability and Substitutability. 4th Intl. Workshop on Symmetry and Constraint Satisfaction Problems (SymCon'04), 95-100 (2004)
7. Freuder, E.C., Hubbe, P.D.: Extracting Constraint Satisfaction Subproblems. IJCAI 1995, 548-557 (1995)
8. Freuder, E.C., Hubbe, P.D.: Using Inferred Disjunctive Constraints To Decompose Constraint Satisfaction Problems. IJCAI 1993: 254-261 (1993)
9. Mehta, D., O'Sullivan, B., Quesada, L., Wilson, N.: Search Space Extraction. CP 2009, Springer LNCS 5732, 608-622 (2009)
10. Ginsberg, M.L., Harvey, W.D.: Iterative Broadening. Artif. Intell. 55(2-3), 367-383 (1992)
11. Freuder, E.C., Synthesizing Constraint Expressions. CACM 21(11), 958-966 (1978)
12. Freuder, E.C., Elfe, C.D.: Neighborhood Inverse Consistency Preprocessing. AAAI 1996, 202-208 (1996)
13. Jeavons, P., Cohen, D., Cooper, M.C.: A substitution operation for constraints. PPCP'94, Springer LNCS 874, 161-177 (1994)