Eliminating Interchangeable Values in Constraint Satisfaction Problems

Eugene C. Freuder

Department of Computer Science
University of New Hampshire
Durham, NH 03824
ecf@cs.unh.edu

Abstract

Constraint satisfaction problems (CSPs) involve finding values for variables subject to constraints on which combinations of values are permitted. This paper develops a concept of *interchangeability* of CSP values. Fully interchangeable values can be substituted for one another in solutions to the problem. Removing all but one of a set of fully interchangeable values can simplify the search space for the problem without effectively losing solutions. Refinements of the interchangeability concept extend its applicability. Basic properties of interchangeablity and complexity parameters are established. A hierarchy of local interchangeability is defined that permits recognition of some interchangeable values with polynomial time local computation. Computing local interchangeability at any level in this hierarchy to remove values before backtrack search is guaranteed to be cost effective for some CSPs. Several forms of weak interchangeability are defined that permit eliminating values without losing all solutions. Interchangeability can be introduced by grouping values or variables, and can be recalculated dynamically during search. The idea of interchangeability can be abstracted to encompass any means of recovering the solutions involving one value from the solutions involving another.

Introduction

A solution to a constraint satisfaction problem (CSP) finds values for variables subject to constraints on what combinations of values are permissible. This paper develops a concept of *interchangeability* of CSP values. Interchangeable values will be in a sense redundant. Their removal will simplify the problem space.

Definition: A value b for a CSP variable V is *fully interchangeable* with a value c for V iff

1. every solution to the CSP which contains b remains a solution when c is substituted for b, and
2. every solution to the CSP which contains c remains a solution when b is substituted for c.

In other words the only difference in the sets of solutions involving b and c are b and c themselves. We can replace a set of fully interchangeable values with a single representative of the set without effectively losing any solutions. It is not necessary to retain all the fully interchangeable values during the search process, for solutions involving one can easily be recovered from solutions involving another. I extend this basic insight in a number of directions to make it more useful in practice.

Figure 1 shows a simple graph coloring problem: color the vertices so that no two vertices which are joined by an edge have the same color. The available colors at each vertex are shown. The colors green, maroon, purple, white and yellow, for vertex Y are fully interchangeable. For example, substituting maroon for green in the solution red|X (red for X), green|Y, blue|Z yields another solution red|X, maroon|Y, blue|Z.

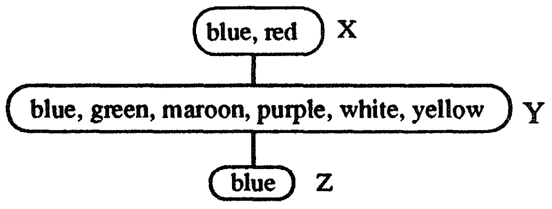

Figure 1. Full interchangeability.

My intuition is that real-world problems may well contain values which are, more or less, interchangeable. In configuration tasks, for example, we may find that, for a particular piece of a particular assembly, several stock parts can serve equally well. In a conventional CSP algorithm needless search effort might be expended on these interchangeable parts. Forms of interchangeability have been used by Van Hentenryck to reduce the search space for car-sequencing and graph-coloring problems (Van Hentenryck 1988, 1989). Other related work can be found in (Yang 1990) and (Mackworth, Mulder, & Havens 1985).

(It may well be that puzzles, like the 8-queens problem, often used to illustrate CSP algorithms, will not benefit greatly from basic interchangeability techniques, that they are puzzles in part because they are unusually particular about which pieces of the puzzle will fit together. However, advanced forms of interchangeability may be still be of use; see the section on functional interchangeability below.)

Interchangeability techniques complement the usual CSP inconsistency methods, which attempt to remove values that will not participate in any solution (Mackworth 1977; Freuder 1978). These techniques can lead to removal of values that may well participate in solutions. It is just that these values succeed (or possibly fail) "equally well". Inconsistency can also be viewed as a special case of interchangeability: inconsistent values all participate in the same set of solutions: the empty set. Removing interchangeable values complements work on removing redundant constraints from CSPs (Dechter & Dechter 1987). Interchangeability emphasizes what I call the *microstructure* of a CSP. The microstructure involves the pattern of consistency connections between values as opposed to variables.

Eliminating interchangeable values can prune a great deal of effort from a backtrack search tree. The example of Figure 1 demonstrates this on a very small scale: with variables and values chosen in lexicographic order during search, eliminating redundant interchangeable values results in a backtrack search tree with half as many branches. If we are seeking all solutions, interchangeability allows us to find a family of similar solutions without what might otherwise involve a complete duplication of effort, for each member of the family. The processing necessary to automate the removal of interchangeable values may prove particularly useful in contexts where constraint networks are used as knowledge bases subject to multiple queries, over which such preprocessing can be amortized.

For simplicity I will assume binary CSPs, which involve only constraints between two variables. However, interchangeability clearly applies to non-binary CSPs (and non-binary CSPs can be transformed into binary ones (Rossi, Dhar, & Petrie 1989)).

Section 2 discusses local forms of interchangeability. Section 3 provides means of taking advantage of interchangeability even when such opportunities are not strictly or immediately available. Along the way I introduce a new, concrete methodology for evaluating CSP search enhancements with best and worst case constructions, and suggest that interchangeability can motivate concept formation and problem decomposition.

Local Interchangeability

Completely identifying fully interchangeable values would seem, in general, to require solving for all solutions. This section identifies various forms of local interchangeability that are more tractable computationally.

Neighborhood Interchangeability

This section defines a basic form of local interchangeability, *neighborhood interchangeability*. Neighborhood interchangeability is a sufficient, but not necessary, condition for full interchangeability. All redundancy induced by neighborhood interchangeability can be removed from a CSP in quadratic time.

Consider the coloring problem in Figure 2. Colors red and white for vertex W are interchangeable from the point of view of the immediate neighbors, X and Y. Red and white are both consistent with any choice for X and Y. The blue value for W is different, it obviously is not consistent with a choice of blue for either X or Y.

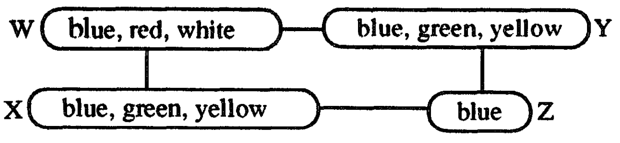

Figure 2. Neighborhood interchangeability.

We will say that red and white are neighborhood interchangeable values for W. The term "neighborhood" is not being used simply because it is motivated by the coloring problem. CSPs are commonly represented by constraint graphs, where vertices correspond to variables and edges to constraints. (Constraint graphs for graph coloring problems conveniently, or confusingly, have the same graph structure as the graph to be colored.) In general we have the following:

Definition: Two values, a and b, for a CSP variable V, are *neighborhood interchangeable* iff for every constraint C on V:

$$\{ i \mid (a,i) \text{ satisfies } C\} = \{ i \mid (b,i) \text{ satisfies } C\}.$$

Notice that the blue value for W is in fact fully interchangeable with red and white, even though it is not neighborhood interchangeable with them. The reason for this is that blue must be chosen for Z, thus cannot be chosen for X or Y. Thus blue for W will fit into any complete solution that red or white will. The incompatibility of blue for W with blue for X and Y does not matter in the end. On the other hand, since red and white are neighborhood interchangeable, there is no way they could fail to be interchangeable in any complete solution: there is no constraint that one could satisfy and not the other.

More generally we have the following simple theorem:

Theorem 1: Neighborhood interchangeability is a sufficient, but not a necessary condition for full interchangeability.

To identify neighborhood interchangeable values we can construct discrimination trees. The leaves of the trees will be the equivalence classes of neighborhood interchangeable values. The process relys on a canonical ordering for variable and value names; without loss of generality we will assume lexicographic ordering.

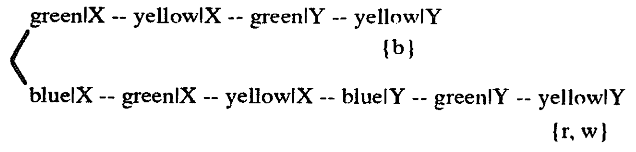
green|X -- yellow|X -- green|Y -- yellow|Y
{b}
blue|X -- green|X -- yellow|X -- blue|Y -- green|Y -- yellow|Y
{r, w}

Figure 3. Discrimination tree.

Figure 3 shows the discrimination tree for variable W. For the blue value we build a branch containing first the consistent values for X, then the consistent values for Y. We do the same for the red value. As we start at the root to build the branch for the white value, we see that the first consistent value, blue for X, is already present as one of the children, so we move down to it. We keep following preexisting branches as long as we can; in this case, we follow one all the way to the end, and red and white are found to be equivalent.

Neighborhood Interchangeability Algorithm:

```
Finds neighborhood interchangeable values in a CSP.
Repeat for each variable:
  Build a discrimination tree by:
  Repeat for each value, v:
    Repeat for each neighboring variable W:
      Repeat for each value w consistent with v:
        Move to if present, construct if not, a node of
        the discrimination tree corresponding to w|W
```

A complexity bound for this algorithm can be found by assigning a worst case bound to each repeat loop. Given n variables, at most d values for a variable, we have the bound (the factors correspond to the repeat loops in top-down order):

$$O(n * d * (n-1) * d) = O(n^2d^2).$$

While this algorithm will find all neighborhood interchangeable values exhaustively, a practitioner might observe some interchangeabilities informally. Semantic groupings can help suggest where to look. For example, in Figure 4, we see a variation on the coloring problem, where the allowable color combinations, e.g. red for X and orange for Y, are indicated by links. (Note that this is unlike the usual constraint graph convention where constraint graph edges represent entire constraints. We are representing the microstructure of the problem here: each link joins an allowable pair of values.) Red and orange are interchangeable, for both X and Y, as are blue and green. Semantic knowledge of the "warm" and "cool" color concepts might suggest looking for such interchangeability.

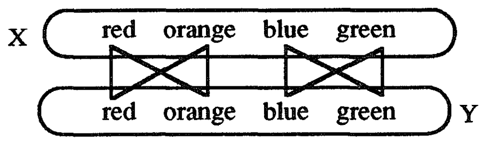

Figure 4. Semantic interchangeability.

On the other hand it might be interesting to view the grouping by an interchangeability algorithm into equivalence classes of red and orange, blue and green, as a *concept formation* process. The "functional" semantics inherent in the underlying problem (perhaps a problem in decoration or design) motivates creation of classes of colors, corresponding to the conventional concepts of warm and cool.

K-interchangeability

This section introduces levels of local interchangeability through a concept called *k-interchangeability*. K-interchangeability involves CSP *subproblems*. For our purposes a subproblem of a CSP will consist of a subset S of the CSP variables along with all the constraints among them, call this the subproblem *induced* by S.

Definition: Two values, a and b, for a CSP variable V, are *k-interchangeable* iff a and b are fully interchangeable in any subproblem of the CSP induced by V and k-1 other variables.

Observe that 2-interchangeability is equivalent to neighborhood interchangeability. (Values will be trivially interchangeable in a subproblem if there are in fact no constraints in the subproblem.) For a problem with n variables, n-interchangeability is equivalent to full interchangeability. (A set is, of course, a subset of itself.) The term *local interchangeability* will be used to refer to k-interchangeability for $k<n$.

The theorem of the previous section generalizes:

Theorem 2. For $i<j$, i-interchangeability is a sufficient, but not necessary condition for j-interchangeability. In particular, any level of local interchangeability is sufficient to ensure full interchangeability.

Proof. As the level of interchangeability increases we can only increase the size of the interchangeability equivalence classes.

A solution to a subproblem may fail to be part of a solution to a larger subproblem, removing an impediment to interchangeability as k increases. Thus the condition is not a necessary one.

It is sufficient, however. Suppose a and b, values for V, are i-interchangeable. I claim they are j-interchangeable. Suppose not. Then there is a j-tuple subproblem solution where substituting a for b (or vice versa) fails to produce another solution. The failure involves at least one constraint, between V and another variable, say U. Throw away j-i elements of the j-tuple solution, but make sure you keep the values for V and U. You now have a solution to an i-tuple of variables where substituting a for b (or vice versa) fails to produce another solution. This contradicts the assumed i-interchangeability. Q.E.D.

The algorithm for finding neighborhood interchangeable values generalizes to an algorithm for finding k-interchangeability. The assumed ordering of the variables and values induces a canonical ordering of variable and value tuples. Each entry in the discrimination net is now a (k-1)-tuple of values.

K-Interchangeability Algorithm:

```
Finds k-interchangeable values in a CSP.
Repeat for each variable:
  Build a discrimination tree by:
  Repeat for each value, v:
    Repeat for each (k-1)-tuple of variables W:
      Repeat for each (k-1)-tuple of values w, which
      together with v constitute a solution to the
      subproblem induced by W:
        Move to if present, construct if not, a node of
        the discrimination tree corresponding to w|W
```

Complexity Analysis

This section is concerned with local interchangeability complexity issues. A complexity bound for the k-interchangeability algorithm is obtained, $O(n^k d^k)$, where d is the maximum number of values (the size of the domain) for any variable. There is reason to believe that this is an optimal upper bound. I prove that for *any* level of local interchangeability there are cases in which preprocessing to remove redundant k-interchangeable values before backtracking, *k-interchangeability preprocessing*, will be cost effective.

The complexity analysis of the k-interchangeability algorithm is similar to that for neighborhood interchangeability, allowing for a worst case $O(n^{k-1})$ (k-1)-tuples of variables and d^{k-1} (k-1)-tuples of values, we get a bound:

$$O(n * d * n^{k-1} * d^{k-1}) = O(n^k d^k).$$

The algorithm includes a brute force search for all the solutions of each subproblem. Performance might be improved by carrying out these searchs more efficiently in advance. On the other hand $O(n^k d^k)$ seems likely to be an optimal worst case bound for finding all the subproblem solutions. Since it is hard to imagine how all k-interchangeable values could be identified without completely solving all these subproblems, $O(n^k d^k)$ seems likely to be a tight worst case bound for this identification. Once the equivalence classes of k-interchangeable values are identified one representative of each class can be retained and the rest of the class declared redundant and removed (within the same time bound obviously).

Despite this potentially costly worst case behavior, I claim that removing redundant interchangeable values can yield great savings in some cases. Furthermore this is true even for large k. For *any* k<n there are problems for which preprocessing to remove redundant k-interchangeable values before backtracking will be cost effective. In fact for any k<n, k-interchangeability preprocessing is *arbitrarily* effective in the sense that *whatever* savings you specify, I can find a CSP for which k-interchangeability preprocessing saves that amount of effort.

The basic observation is that eliminating an interchangeable value prunes the subtree below that value in the backtrack search tree. If all values of a single variable are found to be interchangeable, we have effectively eliminated a level of the backtrack tree; we are left with a maximum of d^{n-1} search tree branches, rather than d^n. If all values of i variables are interchangeable the search tree has at most d^{n-i} branches. If all values of all variables are found to be interchangeable, we are effectively done (anything is a solution). If all values of one variable are interchangeable and they all participate in no solutions, we are effectively done (there is no solution).

Theorem 3. For any number of variables n>2, any k<n, and any computation cost C, there exist CSPs for which the cost of solving by backtrack search exceeds by more than C the cost of solving by k-interchangeability preprocessing followed by backtrack search. This is true regardless of whether we take "solving" to mean finding a single solution, finding all solutions or finding that there is no solution.

Proof. For each interpretation of "solving" I demonstrate that I can construct CSPs with the desired property. I show that the best results we can hope for on those CSPs without preprocessing is worse, by at least C, than the worst behavior we can expect with preprocessing. For simplicity assume that a constructed CSP will have the same number of values, d, for each variable. Assume that backtrack search instantiates variables and chooses values according to their lexicographic ordering.

Consider first the case where there is no solution:

We construct a CSP where failure occurs during backtrack search only when instantiating the last variable, V, and conducting the last consistency check, against variable U. Backtrack search will thus need to examine the complete search space tree. There will be d^n branches in the search tree. Since between most pairs of variables there is really no constraint, I will only count one constraint check per branch. The effort for backtrack search is then $c_1 d^n + c_2$, for appropriate constants c_1 and c_2.

Substituting n-1 into the k-interchangeability bound, we obtain a worst case effort for interchangeability preprocessing of $c_3 n^{n-1} d^{n-1} + c_4$, for appropriate constants c_3 and c_4. The algorithm which identifies (n-1)-interchangeability will discover, and can be trivially altered to report, that all the values of V fail to participate in any solutions to subproblems involving U. This is sufficient to determine that the CSP has no solution, without any subsequent backtrack search.

Now the question is can it be true that:

$$c_1 d^n + c_2 > c_3 n^{n-1} d^{n-1} + c_4 + C\ ?$$

Simple algebra tells us that this will be true if:

$$d > c_5 n^{n-1} + c_6$$

for appropriate constants c_5 and c_6. In other words, we only need construct a problem where the number of values is sufficiently large in comparison with the number of variables.

(The nature of "sufficiently large" may be offputting, but bear in mind this is a worst case scenario for interchangeability involving (n-1)-interchangeability. Also observe that there are simple cases with 2 variables and 3 values that demonstrate the desirability of even (n-1)-interchangeability preprocessing.)

Now consider the case where we search for a single solution:

The construction is similar. This time the last variable in the search tree, V, will be such that k-interchangeability preprocessing reduces the domain of V to a single value. This value will only be consistent with the last value, in the ordering of values, for each other variable. Thus there will be a single solution, which will appear as the "rightmost" branch of the search tree. Again all pairs of values between variables other than V are consistent. We have at most d^{n-1} branches in the search tree. (Actually interchangeability redundancy removal should prune the search tree further.) For each branch there are n-1 non-trivial constraints to check, those between V and the other n-1 variables. Thus the backtrack search effort, after preprocessing reduces the number of variables for V to one, is at most:

$$c_1(n-1)d^{n-1}+c_2.$$

Adding the effort for (n-1)-interchangeability preprocessing we have a total effort of:

$$(c_1(n-1)d^{n-1}+c_2) + (c_3n^{n-1}d^{n-1}+c_4).$$

Without preprocessing the search tree will have $d^n-(d-1)$ branches. Search will repeatedly try all d values for V, until success is achieved with the last value for every other variable. Thus search effort will be:

$$c_5(n-1)(d^n-(d-1))+c_6.$$

We want:

$$(c_1(n-1)d^{n-1}+c_2)+(c_3n^{n-1}d^{n-1}+c_4)$$
$$< c_5(n-1)(d^n-(d-1))+c_6+C.$$

Simplifying, it is sufficient that:

$$d > c_7n^{n-1} + c_8$$

for appropriate constants c_7 and c_8.

Finally, consider the case where we are looking for all solutions:

This time construct a CSP where k-interchangeability reduces a variable V to a single value and that variable is the first variable to be instantiated in the backtrack search order. Consider the final variable, W. Arrange that only the final value for W is consistent with anything, and it is only consistent with the final value for each of the other variables (with the exception of V; it is consistent with all values of V). Other pairs of values are all consistent. Thus backtrack search will check out d^{n-1} branches before finding the first solution. Backtrack search after interchangeability preprocessing will require if anything less effort to reach the first solution (interchangeability can reduce the number of values for other variables). Once the first solution has been found interchangeability preprocessing will permit simply substituting to obtain the other d-1 solutions, while backtrack search requires searching another size d^{n-1} search tree for each additional solution. Clearly, for sufficiently large d, the savings here can be as large as we like. Q.E.D.

Observe that the constructions in the proof work for any level of k-interchangeability, including the lowest. If 2-consistency, for example, is sufficient to create a situation like those constructed above, the savings can be dramatic, even without an inflated value for d, and not just for finding multiple solutions. It may be that the potential payoff from the relatively inexpensive 2-interchangeability preprocessing--or the potential cost of doing without it--will motivate routine preprocessing for 2-interchangeability.

Theorem 4: For any n>2 and any d, there exist CSPs with n variables and d values for each, for which the savings achieved by preprocessing for 2-interchangeability is $O(n^2d^n)$.

Proof. Construct a CSP with no solution where none of the values for one variable, V, are consistent with any value for one of the others, U, while all other value pairs are permitted. 2-interchangeability will discover that there is no solution with $O(d^2n^2)$ effort. Assume a search order where U is the first variable to be instantiated and V the last. This will require a full backtrack tree search with $O(n^2d^n)$ effort. The difference is $O(n^2d^n)$. Q.E.D.

Observe further that the cost of (n-1)-interchangeability preprocessing is such that the arguments in Theorem 3 focused not on how easily we could proceed with the search after removing redundant values, but on ensuring that backtrack search without preprocessing would require a sufficiently large effort. The calculations did not take into account that interchangeability can affect more than a single variable. Even 2-interchangeability can significantly reduce the number of values for many of the variables (in the extreme down to a single value for each variable) resulting in a major savings in the effort required to find one, or all, solutions.

Note that very similar arguments to those in this section should produce similar results regarding the efficacy of k-consistency preprocessing. Indeed constructions in this section are reminiscent of the basic "thrashing" arguments that long ago pointed out problems with conventional backtrack search, motivating consistency "relaxation" preprocessing techniques among other refinements (Bobrow & Raphael 1974). However, I do not believe that this kind of concrete analysis has been carried out previously for thrashing type behavior.

Extended Interchangeability

This section provides means of taking advantage of interchangeability even when such opportunities are not strictly or immediately available.

Weak Interchangeability

This section defines several forms of *weak interchangeability*. These may involve sacrificing some solutions, but this will not matter if we are seeking a single solution. Locally computable forms of these concepts are available.

Substitutability. The simplest form of weak interchangability is substitutability; this captures the idea that interchangeability can be restricted to a "one-way" concept.

Definition: Given two values, a and b, for a CSP variable V, a is *substitutable* for b iff substituting a in any solution involving b yields another solution.

We can remove b from the problem, knowing that we have not removed all the solutions. If there was any solution involving b, there will remain a solution where a is substituted for b. However we cannot recover solutions involving b by substituting b in the solutions involving a, as we do not know which, if any, of those substitutions produce solutions.

If each of two values can be substituted for the other, the two values are fully interchangeable. Substitutability can be computed locally. In particular, we have:

Definition: For two values, a and b, for a CSP variable V, a is *neighborhood substitutable* for b iff for every constraint C on V:

$$\{ i \mid (a,i) \text{ satisfies } C\} \supseteq \{ i \mid (b,i) \text{ satisfies } C\}.$$

In the example of Figure 2, red is neighborhood substitutable for blue for variable W, even though red and blue are not neighborhood interchangeable.

Partial Interchangeability. Partial interchangeability captures the idea that values for variables may "differ" among themselves, but be fully interchangeable with respect to the rest of the world.

Definition: Two values are *partially interchangeable*, with respect to a subset S of variables, iff any solution involving one implies a solution involving the other with possibly different values for S.

When S is the empty set, the values are fully interchangeable. Figure 5 presents an example of partial interchangeability: blue and red for W are partially interchangeable, with respect to the set {X}. Note: blue and red for W are interchangeable as far as V is concerned; blue for W goes with red for X, and red for W with blue for X; blue and red for X are interchangeable as far as Y and Z are concerned. Thus while substituting say red for blue in a solution for W necessitates a change in the value for X, it will not require any change in the values for V, Y or Z.

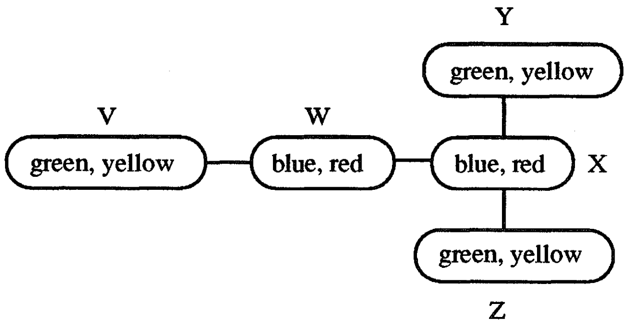

Figure 5. Partial interchangeability.

Subproblem Interchangeability. Subproblem interchangeability captures the idea that values can be interchangeable within a subproblem of the CSP. Subproblem interchangeability may motivate and guide a divide and conquer decomposition of a CSP.

Definition: Two values are *subproblem interchangeable*, with respect to a subset of variables S, iff they are fully interchangeable with regards to the solutions of the subproblem of the CSP induced by S.

Note the values are required to be fully interchangeable with regard to the subproblem, not the complete CSP. Of course, when S is the entire set of variables, the subproblem is the complete CSP, and the values are fully interchangeable for the CSP. Subproblem interchangeability and partial interchangeability are not quite inverse notions.

Theorem 5: Subproblem interchangeability with respect to S implies partial interchangeability with respect to S', the variables not in S; however, partial interchangeability with respect to S does not imply subproblem interchangeability with respect to S'.

Proof: The key observation is that a solution to a subproblem may fail to appear as a portion of any solution to the complete problem. On the other hand if we take from a solution to the complete CSP the values for a subset of variables, those values will constitute a solution to the subproblem induced by those variables.Q.E.D.

Meta-interchangeability

By grouping variables into "metavariables", or values into "metavalues", we can introduce interchangeability into higher level "metaproblem" representations of the original CSP. Meta-interchangeability might also be viewed as providing motivation and guidance for dividing CSP variables into subproblems and CSP values into concept hierarchies (Mackworth, Mulder, & Havens 1985).

Figure 6 presents an example of metavalue grouping. As in Figure 4 we indicate the allowable pairs of values in the original CSP with links, e.g. yellow for Y is consistent with light blue and light red for X. In the original problem yellow and brown for Y are not interchangeable. However, if we were to combine sky, light and dark blue into a metavalue "blue", and similarly create the metavalue "red", we would have a problem in which yellow and brown are fully interchangeable.

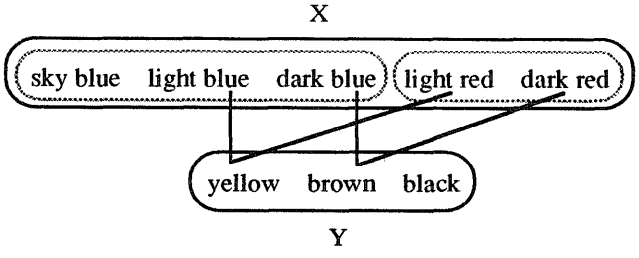

Figure 6. Meta-interchangeability.

We can also merge variables into metavariables. The values of a metavariable will be the solutions of the subproblem induced by the individual variables. Values for two metavariables will be consistent if the component values for the original variables are all consistent. Forming the metaproblem may create new interchangeabilities.

Dynamic Interchangeability

Interchangeability can be recalculated after choices are made for variable values during backtrack search. It can be recalculated after inconsistent values have been filtered out during the search process in a preprocessing step or by a "hybrid" algorithm that interleaves backtracking and relaxation. Interchangeability can also be recalculated to reflect changes in a dynamic constraint representation. Interchangeability might be sought dynamically during a knowledge acquisition or problem definition process.

The idea of integrating local interchangeability recalculations with backtrack search is especially intriguing given the success of local consistency calculations in enhancing backtrack search performance.

Functional Interchangeability

The essential idea of interchangeability is that given the solutions involving one value, we can recover the solutions involving another. We have been using simple substitution to go from one set of solutions to another. However, substitution is only the simplest function we could use.

Definition: Let $S_{v|V}$ be the set of solutions inclusing value v for variable V. CSP values a for V and b for W are *functionally interchangeable* iff there exist functions f_a and f_b such that $f_a(S_{a|V}) = S_{b|W}$ and $f_b(S_{b|W}) = S_{a|V}$. (V and W may be the same variable.)

This is a very general definition that deserves further study. The definition does not even require that a and b be values for the same variable. In fact, strictly speaking, any two values are functionally interchangeable; once we have all the solutions we can give a "brute force" definition of the necessary functions. The key obviously is for the functions to be a priori available or cost effective to obtain.

One natural refinement of the definition involves a solution preserving function on variable values:

Definition: Two values, a and b, for a CSP variable, are *isomorphically interchangeable* iff there exists a 1-1 function f such that:

1. $b = f(a)$
2. for any solution S involving a, $\{f(v) \mid v \in S\}$ is a solution
3. for any solution S involving b, $\{f^{-1}(v) \mid v \in S\}$ is a solution.

Problem symmetry is a likely source of this sort of transformational interchangeability. Consider the 8-queens problem, for example, placing 8 queens on a chessboard such that no two attack one another, where rows correspond to variables and columns to values. The reduction of the values in the first row suggested in (Reingold, Nievergelt, & Deo 1977) can be viewed as an application of isomorphic interchangeability.

First observe that the column 1 position for the queen in the first row is isomorphically interchangeable with the column 8 position, the 2 position with the 7 position, etc. The interchangeability function f maps position i into position 9-i, for each row, simulating flipping the board about its vertical axis of symmetry. This permits eliminating positions 5 through 8 for the first row. Next observe that because of symmetry about the diagonal axes position 1 of row 1 is isomorphically interchangeable with position 8 of row 8, thus we can eliminate position 1 for the first row.

Acknowledgments. This material is based upon work supported by the National Science Foundation under Grant No. IRI-8913040. The Government has certain rights in this material. The author is currently a Visiting Scientist at the MIT Artificial Intelligence Laboratory.

References

(Bobrow & Raphael 1974) New programming languages for artificial intelligence research. *Comput. Surv. 6*,3, 153-174.

(Dechter & Dechter 1987) Removing redundancies in constraint networks. *Proc. AAAI-87*, 105-109.

(Freuder 1978) Synthesizing constraint expressions. *Commun. ACM 21*,11, 958-966.

(Mackworth 1977) Consistency in networks of relations. *Artif. Intell. 8*, 99-118.

(Mackworth, Mulder, & Havens 1985) Hierarchical arc consistency: exploiting structured domains in constraint satisfaction problems. *Comput. Intell. 1*, 118-126.

(Reingold, Nievergelt, & Deo 1977) *Combinatorial Algorithms*. Prentice-Hall.

(Rossi, Dhar, & Petrie 1989) On the equivalence of constraint satisfaction problems. MCC Technical Report ACT-AI-222-89. MCC, Austin, Texas 78759.

(Van Hentenryck 1988) Solving the car-sequencing problem in constraint logic programming, *Proc. ECAI-88*.

(Van Hentenryck 1989) A logic language for combinatorial optimization, *Annals of Operations Research 21*, 247-274.

(Yang 1990) An algebraic approach to conflict resolution in planning. *Proc. AAAI-90*, 40-45.