Domain Transmutation in Constraint Satisfaction Problems[*]

James Bowen and **Chavalit Likitvivatanavong**

Cork Constraint Computation Centre
University College Cork, Ireland
{j.bowen, chavalit}@4c.ucc.ie

Abstract

We study local interchangeability of values in constraint networks based on a new approach where a single value in the domain of a variable can be treated as a combination of "sub-values". We present an algorithm for breaking up values and combining identical fragments. Experimental results show that the transformed problems take less time to solve for all solutions and yield more compactly-representable, but equivalent, solution sets. We obtain new theoretical results on context dependent interchangeability and full interchangeability, and suggest some other applications.
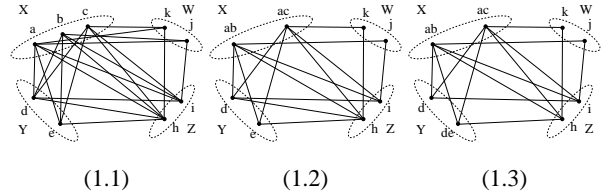
Introduction

Interchangeability of domain values was first reported by (Freuder 1991), where neighborhood interchangeability (NI) and neighborhood substitutability (NS) are identified as local properties that can be exploited to remove redundant values, either as a preprocessing step or during search (Benson & Freuder 1992; Haselböck 1993). NI was later extended to cover interchangeable values in different variables (Choueiry & Noubir 1998).

We present a new approach to local interchangeability in this paper. Normally, each domain value is treated as atomic. Our idea is to "split the atom" — a domain value can be split into several "sub-values" (Figure (2.1),(2.2)) so that interchangeable fragments from other values can then be merged to avoid duplicate search effort during the solving process.

Figure (1.1),(1.2),(1.3) illustrate. The problems depicted are in microstructure form: each edge connects compatible values in the domains of two variables. Figure (1.1) shows the original problem, while (1.2) shows the result of splitting and merging sub-atomic value fragments in the domain of X (a process which, by analogy with Nuclear Chemistry,[1] we call *domain transmutation*). Figure (1.3) shows the result of subsequently transmutating the domain of Y.

Consider, for example, the values a, b and c shown in (1.1). Value a supports the following tuples in $Y \times Z \times W$:

[1]Transmutation — a change in the number of protons in the nucleus of an atom, producing an atom with a different atomic number.

(1.1)　　　　　(1.2)　　　　　(1.3)

(d,h,j), (d,h,k), (d,i,j), (d,i,k), (e,h,j), (e,h,k), (e,i,j) and (e,i,k). We denote this support relationship concisely by $(\{\boldsymbol{a}\}, \{d,e\} \times \{h,i\} \times \{j,k\})$. The supports of b and c are $(\{\boldsymbol{b}\}, \{d,e\} \times \{h,i\} \times \{j\})$ and $(\{\boldsymbol{c}\}, \{d,e\} \times \{h,i\} \times \{k\})$. No two of the values a, b and c are NI. However, a is neighborhood-substitutable for b and also for c.

The value a can be split into two fragments: $(\{\boldsymbol{a}\}, \{d,e\} \times \{h,i\} \times \{j\})$ and $(\{\boldsymbol{a}\}, \{d,e\} \times \{h,i\} \times \{k\})$. The first fragment of a is interchangeable with b, while the second fragment of a is interchangeable with c. Instead of eliminating one of the interchangeable fragments, we merge them together and use their combined labels as the new value's label, to indicate its origin. Thus, after merging we have two values: $(\{\boldsymbol{a},\boldsymbol{b}\}, \{d,e\} \times \{h,i\} \times \{j\})$ and $(\{\boldsymbol{a},\boldsymbol{c}\}, \{d,e\} \times \{h,i\} \times \{k\})$, depicted as ab and ac in (1.2).

Similarly, consider the values d and e in (1.2). The supports of d and e are $(\{\boldsymbol{d}\}, \{ab,ac\} \times \{h,i\})$ and $(\{\boldsymbol{e}\}, \{ab,ac\} \times \{h\})$. However, d can be split into two fragments: $(\{\boldsymbol{d}\}, \{ab,ac\} \times \{h\})$ and $(\{\boldsymbol{d}\}, \{ab,ac\} \times \{i\})$. The first fragment is interchangeable with e. The result is (1.3). No two values from the same domain intersect in this problem.

As demonstrated, our approach subsumes both NI and NS, but it does so by applying the NI principle to value-*fragments*. Indeed, while standard NS cannot preserve the solution set of a CSP, our approach can, precisely because it treats value-substitutability as fragment-interchangeability. For example, the solution (ac, de, h, k) in (1.3) corresponds to four solutions in the original problem: (a,d,h,k), (a,e,h,k) (c,d,h,k), and (c,e,h,k).

We will show empirically that domain transmutation can reduce the amount of time needed to find all solutions, as well as enable more compact representations of solution sets. This is essential in certain interactive constraint-based applications, such as interactive design-advice systems (O'Sullivan 2002): sometimes, to make his next interactive decision, a design engineer needs to be able to examine *all* consistent assignments to the variables involved

in a sub-region of the overall CSP, and to see how similar they are to each other. Another application involves compiling CSPs (Weigel & Faltings 1999): all solutions to a sub-problem are required in order to create a meta-variable, after which NI is employed to reduce domain size.

Background

We focus on binary CSPs since non-binary constraints can be converted into binary constraints.

Definition 1 (Binary CSP) A *binary CSP* $\mathcal{P}$ is a triplet $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ where $\mathcal{V}$ is the finite set of variables, $\mathcal{D} = \bigcup_{V \in \mathcal{V}} D_V$ where D_V is the finite set of possible values for V (sometimes we denote D_V by $\mathcal{D}(V)$), and $\mathcal{C}$ is a finite set of constraints such that each $C_{XY} \in \mathcal{C}$ is a subset of $D_X \times D_Y$ indicating the compatible pairs of values for X and Y, where $X \neq Y$. If $C_{XY} \in \mathcal{C}$, then $C_{YX} = \{(y, x) \mid (x, y) \in C_{XY}\}$ is also in $\mathcal{C}$. The *neighborhood* of variable V, denoted by N_V, is the set $\{W \mid C_{VW} \in \mathcal{C}\}$.

A *partial assignment* is a function $\pi : \mathcal{W} \subseteq \mathcal{V} \to \mathcal{D}$ such that $\pi(W) \in D_W$ for all $W \in \mathcal{W}$; we denote the function domain by *dom(π)*. π is *consistent* iff for all $C_{XY} \in \mathcal{C}$ such that $X, Y \in dom(\pi)$, $(\pi(X), \pi(Y)) \in C_{XY}$; we denote consistency of π by *cons(π)*. π is a *solution* iff $dom(\pi) = \mathcal{V}$ and *cons(π)*. The set of all solutions for $\mathcal{P}$ is denoted by *Sols($\mathcal{P}$)*. We use $\mathcal{P}|_{\mathcal{W}}$ to denote the CSP induced by $\mathcal{W} \subseteq \mathcal{V}$.

Given a partial assignment π, we define $\pi[a/b]$ to be the partial assignment where $\pi[a/b](V) = b$ if $\pi(V) = a$; otherwise $\pi[a/b](V) = \pi(V)$.

At this point we extend the usual definition of a value to a 2-tuple in order to clearly distinguish between its syntax (label) and semantics (support). This lower-level detail is only of theoretical concern and can be safely ignored in other contexts.

Definition 2 (Values) A *value* a is a 2-tuple (L, σ) where L is a set of *labels*, while σ, called *support*, is a function $\sigma : N_V \to 2^{\mathcal{D}}$ such that $\sigma(W) \subseteq D_W$ for any $W \in N_V$. We use L_a to denote the set of labels of a and use σ_a to denote the support of a. A value a in $\mathcal{D}$ of the CSP $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ must be *valid*, that is, $\sigma_a(W) = \{b \in D_W \mid (a, b) \in C_{VW}\}$ for any $W \in N_V$.

Let $\mathcal{L} = \bigcup_{a \in \mathcal{D}} L_a$ be the set of all labels. A *partial label assignment* is a function $\lambda : \mathcal{W} \subseteq \mathcal{V} \to \mathcal{L}$ such that $\lambda(W) \in \bigcup_{a \in D_W} L_a$ for all $W \in \mathcal{W}$. Given a partial assignment π, we denote π_{label} to be the set of partial label assignments $\{\lambda \mid \lambda(V) \in L_{\pi(V)}\}$. For any set of partial assignments $\mathcal{X}$, we use $\mathcal{X}_{label}$ to denote $\bigcup_{\pi \in \mathcal{X}} \pi_{label}$.

The new definition allows a variable domain to contain values having the same label but differrent supports (e.g. Figure (2.2)). Note that NI is still allowed: it is possible to have values with the same supports but different labels. However, values having both the same labels and supports are not permitted. In practice, when structure can no longer be used to differentiate values, each label can still be made unique and yet indicate its origin simply by appending to it

some suffix. (e.g. r_1 and r_2 in Figure (3.2)). A solution can be converted back to the original format in time $O(|\mathcal{V}|)$.

Each value can have multiple labels so that we can combine NI values without losing any solution (unlike the standard practice where only one value is kept). For example $(\{\boldsymbol{a}\}, \{x\} \times \{y\})$ and $(\{\boldsymbol{b}\}, \{x\} \times \{y\})$ can be replaced by a single value $(\{\boldsymbol{a},\boldsymbol{b}\}, \{x\} \times \{y\})$. Any solution π involving $(\{\boldsymbol{a},\boldsymbol{b}\}, \{x\} \times \{y\})$ can be converted using π_{label}. Throughout this paper, we always assume CSPs with no NI values since NI values are trivially handled by this approach.

Given a CSP we define the following operations on values, analogous to the usual operations on sets.

Definition 3 (Operations on Values) Let a and b be two values in D_V.

The *intersection* of a and b (denoted by $a \odot b$) is a value c where $L_c = L_a \cup L_b$ and $\sigma_c(W) = \sigma_a(W) \cap \sigma_b(W)$ for all $W \in N_V$. The intersection is undefined (denoted by $a \odot b = \emptyset$) if there exists a variable $X \in N_V$ such that $\sigma_a(X) \cap \sigma_b(X) = \emptyset$. Two values a and b, are said to be *disjoint* if $a \odot b = \emptyset$. A set of values is disjoint if its members are pairwise disjoint.

The *union* of a and b (denoted by $a \oplus b$) is a value c where $L_c = L_a$ and $\sigma_c(W) = \sigma_a(W) \cup \sigma_b(W)$ for all $W \in N_V$. The union is undefined (denoted by $a \oplus b = \emptyset$) if $L_a \neq L_b$ or there exist two or more variables $X \in N_V$ such that $\sigma_a(X) \neq \sigma_b(X)$. The *subtraction* of b from a (denoted by $a \ominus b$) is the smallest set[2] of disjoint values $\mathscr{C}$ such that $\bigoplus(\mathscr{C} \cup \{a \odot b\}) = a$.

Value a is *subsumed*[3] by b (denoted by $a \sqsubseteq b$) if $\sigma_a(W) \subseteq \sigma_b(W)$ for all $W \in N_V$.

We will continue to use cross-product representation to denote supports for a value. For instance, consider values $(\{\boldsymbol{a}\}, \{d,e\} \times \{h,i\} \times \{j,k\})$ and $(\{\boldsymbol{b}\}, \{d,e\} \times \{h,i\} \times \{j\})$, which are depicted by their labels a and b in Figure (1.1). Their intersection is $(\{\boldsymbol{a},\boldsymbol{b}\}, \{d,e\} \times \{h,i\} \times \{j\})$. Another example: $(\{\boldsymbol{x},\boldsymbol{y}\}, \{a,b\} \times \{c\}) \odot (\{\boldsymbol{y},\boldsymbol{z}\}, \{a\} \times \{c,d\}) = (\{\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}\}, \{a\} \times \{c\})$.

It should be emphasized that the result of subtraction may be a *set* of values. Consider $(\{\boldsymbol{a}\}, \{d,e\} \times \{f,g\}) \ominus (\{\boldsymbol{b}\}, \{d\} \times \{f\})$. The result is a set $\{(\{\boldsymbol{a}\}, \{e\} \times \{f,g\}), (\{\boldsymbol{a}\}, \{d\} \times \{g\})\}$. (This kind of fragmentation is not uncommon. On a larger scale, extracting subproblems can also fracture the remainder of a CSP (Freuder & Hubbe 1995).)

Lemma 1 Let a and b be values in D_V and let $W \in N_V$.
(1) Let $S \subseteq D_W$, then $S \subseteq \sigma_a(W)$ iff $a \in \bigcap_{d \in S} \sigma_d(V)$.
(2) $\mathscr{C} = \{a \odot b\} \cup (a \ominus b) \cup (b \ominus a)$ is disjoint[4]. Moreover, $\{\pi \mid cons(\pi) \wedge (dom(\pi) = \{V\} \cup N_V) \wedge (\pi(V) \in \{a, b\})\}_{label} = \{\pi \mid cons(\pi) \wedge (dom(\pi) = \{V\} \cup N_V) \wedge (\pi(V) \in \mathscr{C})\}_{label}$.
(3) $\odot$ and $\oplus$ are idempotent, commutative, and associative.

We now formally define domain transmutation.

---

[2] Subtraction is unique, although we do not have enough space to provide the proof.

[3] a is neighborhood substitutable for b if and only if $b \sqsubseteq a$.

[4] Recall that the subtraction of two values may result in a collection of values.

Definition 4 (Transmutation) Given a CSP $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$. For any $V \in \mathcal{V}$, a *transmutation* of D_V (denoted by $trans(D_V)$) is a variable domain D'_V such that,
(1) D'_V is disjoint.
(2) $Sols(\mathcal{P}|_{\{V\} \cup N_V})_{label} = Sols(\mathcal{P}'|_{\{V\} \cup N_V})_{label}$, where $\mathcal{P}'$ is the CSP in which D_V is replaced by D'_V and all constraints involving V have been updated[5] by values in D_V. We denote $\mathcal{P}'$ by $trans(\mathcal{P}, V)$.

In a domain transmutation, no two values overlap and all the solutions involving the original domain are preserved. An algorithm for transmutating a domain will be given in the next section. Effects of domain transmutation are shown in the following theorem, whose proof is omitted.

Theorem 1 (Transmutation Effects) Given a CSP $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, let $trans(\mathcal{P}, V) = (\mathcal{V}, \mathcal{D}', \mathcal{C}')$,
(1) If $b \in \mathcal{D}'(V)$, then there exists some $a \in \mathcal{D}(V)$ such that $b \sqsubseteq a$ and such that, for any solution π for $trans(\mathcal{P}, V)$ where $\pi(V) = b$, there also exists the solution $\pi[b/a]$ for $\mathcal{P}$.
(2) If π is a solution for $\mathcal{P}$ such that $\pi(V) = a$, then there is *exactly one* $c \in \mathcal{D}'(V)$, $c \sqsubseteq a$, such that $\pi[a/c]$ is a solution for $trans(\mathcal{P}, V)$.

Proposition 1 (Upper Bound on Search Space) Given a CSP $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, we denote *maxassign($\mathcal{P}$)* to be the number of possible partial assignments π such that *dom(π)* $= \mathcal{V}$. For any binary CSP $\mathcal{P}$, *maxassign(trans($\mathcal{P}, V$))* $\leqslant$ *maxassign($\mathcal{P}$)* for all $V \in \mathcal{V}$.

In the rest of this section, we will obtain new results on other types of interchangeability based on transmutation.

Context dependent interchangeability (CDI) was proposed in (Weigel, Faltings, & Choueiry 1996) in an attempt to capture interchangeability in a limited situation. CDI is rephrased as follows.

Definition 5 (Context Dependent Interchangeability)
Values a and b in D_V are *context dependent interchangeable* iff there exist two solutions π_1 and π_2 such that $\pi_1(V) = a$ and $\pi_2 = \pi_1[a/b]$.

Theorem 2 (Intersection of CDI Values) Values a and b in D_V are CDI for a CSP $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ iff in $trans(\mathcal{P}, V)$ $= (\mathcal{V}, \mathcal{D}', \mathcal{C}')$ there is *exactly one* $c \in \mathcal{D}'(V)$ such that $c \sqsubseteq a$ and $c \sqsubseteq b$ and there exists a solution π such that $\pi(V) = c$.

Proof ($\Rightarrow$) From Theorem 1(2) we have c_1 and c_2 and two solutions $\pi[a/c_1]$ and $\pi[a/c_2]$ $(= \pi[a/b][b/c_2])$ for $trans(\mathcal{P}, V)$. But c_1 and c_2 must be the same value because otherwise $c_1 \odot c_2 \neq \emptyset$ $(\pi(W) \in c_1(W) \cap c_2(W)$ for all $W \in N_V)$, contradicting Definition 4(1).
($\Leftarrow$) Given a solution π in $trans(\mathcal{P}, V)$ such that $\pi(V) = c$ where $c \sqsubseteq a$ and $c \sqsubseteq b$, then $\pi[c/a]$ and $\pi[c/b]$ are solutions of $\mathcal{P}$ according to Theorem 1(1). $\square$

Collorary 1 (Necessary Condition for CDI) If values a and b in D_V are CDI then $a \odot b \neq \emptyset$.

[5]To make values in D'_V valid.

Theorem 2 tells us that two values are CDI iff both share the same sub-value and that sub-value is part of a solution. As a result, CDI can be seen as a local condition followed by a global condition. In (Weigel, Faltings, & Choueiry 1996), however, CDI is viewed as a single global condition and has to be detected indirectly by a non-CSP method.

We can exploit this property to find CDI values by transmutating a domain and then solving the resulting problem *independently using any CSP algorithm*. E.g., to check whether a and b in Figure (1.1) are CDI, one only needs to check whether ab in (1.3) is involved in any solution. If two values do not intersect, such as h and i in (1.1), we can tell immediately that they are not CDI according to Corollary 1.

We next consider the relation between full interchangeability (FI) (Freuder 1991) and transmutation. (Freuder 1991) showed that NI implies FI. But what happens when values are FI but not NI? Theorem 3 gives the answer.

Definition 6 (Full Interchangeability) Two values a and b in D_V are FI iff for any solution π,
(1) if $\pi(V) = a$ then $\pi[a/b]$ is a solution.
(2) if $\pi(V) = b$ then $\pi[b/a]$ is a solution.

In contrast to FI, CDI has a weaker requirement in that values do not need to be interchangeable in all solutions — just some of them. On the other hand, FI values do not need to be involved in any solution, whereas it is necessary for CDI.

Theorem 3 (FI Pruning) Two values a and b in D_V are FI iff for any solution π, $(\pi(V) \in \{a,b\}) \Rightarrow (\pi(W) \in \sigma_{a \odot b}(W)$ for all $W \in N_V)$.

Sketch of a Proof: Contrapositive of the fact: there exists a solution π such that $\pi(V) = a$ but $\pi[a/b]$ is not a solution iff there exists $W \in N_V$ such that $\pi(W) \notin \sigma_{a \odot b}(W)$. $\square$

Collorary 2 (Necessary Condition for FI) If $a \odot b = \emptyset$ then a and b are FI iff neither takes part in a solution.

The common way of finding FI values is to look for two values that can be interchangeable in the solution sets. We will show how to use Theorem 3 to reduce the search effort.

Consider the CSP in (3.1); suppose that there is an extra constraint between V_1 and V_3, whose only nogoods are $\{(c,f),(d,h)\}$. Suppose we are interested in finding FI values in V_2, whose domain is $\{r,g,b,x,y,z\}$, and after transmutation becomes what is shown in (3.2). The potential FI pairs of values are r-g, g-b, r-b, and x-y. Although any pair of values that do not intersect can still trivially be FI, this is uninteresting since according to Corollary 2 both of them must not take part in any solution; thus interchangeability of z can be ruled out because it does not intersect with other values.

We can remove rg, gb, and rb from the domain since their involvement in solutions does not affect FI. The domain is now $\{r_1, r_2, g_1, g_2, b_1, b_2, x_1, y_1\}$.

To find all the solutions, suppose we try to instantiate V_2 in the order $(r_1, r_2, g_1, g_2, b_1, b_2, x_1, y_1)$. The first solution involves r_1. According to Theorem 3, r cannot be FI with any other values and so we can rule out r-g and r-b. Moreover,

(2.1)   (2.2)   (3.1)   (3.2)   (3.3)
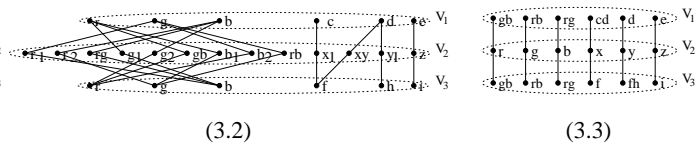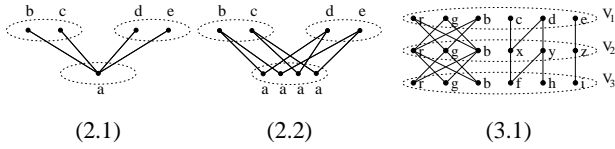
r_2 can be removed from the domain since its involvement in a solution does not give us any new information.

Next we try g_1, and it too takes part in a solution. By the same reasoning, we infer that g cannot be FI with any other values, and so we can prune g_2. We are now done with FI regarding $\{r,g,b\}$ — no FI value involving one of them exists — and so values b_1 and b_2 can be removed.

We carry on with values x_1 and y_1 that are left in the domain; neither is involved in a solution. There is no value left and by Theorem 3, we conclude that x and y are FI.

Domain Transmutation Procedure

The procedure for transmutating the domain of a single variable is stated as pseudo-code below.

Procedure `transmutate`(V)

0 input $\leftarrow D_V$;
0 output $\leftarrow \emptyset$;
0 **while** *input* $\neq \emptyset$ **do**
0   Select and delete any a from input ;
0   match $\leftarrow$ **false** ;
0   **while** *(not match)* **and** *(next b in output exists)* **do**
0   **if** $a \odot b \neq \emptyset$ **then**
0   match $\leftarrow$ **true** ;
0              $\mathscr{A} \leftarrow a \ominus b$;
0              $\mathscr{B} \leftarrow b \ominus a$;
0   Replace b in output with $a \odot b$ and all values in $\mathscr{B}$;
0   Put all values in $\mathscr{A}$ back to input ;

0   **if** *(not match)* **then**
0   Insert a to output
0 Where possible, union values in output ;
0 $D_V \leftarrow$ output ;
0 Update all constraints connecting V;

Theorem 4 (Correctness) Algorithm `transmutate` produces a domain transmutation.

Proof At the end of the algorithm output is a disjoint set of values by induction on the size of output and Lemma 1(2). Since the algorithm depends only on $\odot$ and $\ominus$ to split values, all solutions are preserved by Lemma 1(2). Therefore output satisfies Definition 4 □

The following theorem shows that each variable needs to be transmutated by the algorithm at most once.

Theorem 5 (Transmutation Quiescence) If a variable domain D_V is a transmutation, any subsequent transmutation attempt using the algorithm will result in no change.

Proof This is obviously true if no variable domain in N_V is altered. Otherwise, we will show that after D_V has been transmutated, it remains disjoint regardless of any domain transmutation in N_V.

Let $\mathcal{P}$ be the CSP $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ just after D_V has been transmutated. Let a and b be values in $\mathcal{D}(V)$. Both are disjoint according to Definition 4(1) and so there must be a variable $W \in N_V$ such that $\sigma_a(W) \cap \sigma_b(W) = \emptyset$. Consider *trans*$(\mathcal{P}, W) = (\mathcal{V}, \mathcal{D}', \mathcal{C}')$ and let us suppose that $c \in$

$\sigma_a(W) \cap \sigma_b(W) \neq \emptyset$ as a result of transmutating W, for some value $c \in \mathcal{D}'(W)$.

By Lemma 1(1), $\{a, b\} \subseteq \sigma_c(V)$. Since $c \in \mathcal{D}'(W)$, by Theorem 1(1) $c \sqsubseteq d$ for some $d \in \mathcal{D}(W)$. That means $\{a, b\} \subseteq \sigma_c(V) \subseteq \sigma_d(V)$. By Lemma 1(1), $d \in \sigma_a(W) \cap \sigma_b(W)$ — contradicting $\sigma_a(W) \cap \sigma_b(W) = \emptyset$ for $\mathcal{P}$.

Hence the assumption is false and $\sigma_a(W) \cap \sigma_b(W)$ remains empty after W is transmutated. □

Note that union in line 15 is used in re-combining remaining fragmented values from the same source, although the smallest possible domain transmutation is not required by Definition 4.

During transmutation the size of output could be very large if there many fragments of values left after intersection. If so, it is very unlikely that the values inside can be combined so that the resulting domain is smaller than the original. Therefore we use the *transmutation cutoff heuristic* which imposes an upper bound on the size of output; if it exceeds the upper bound, transmutation is aborted.
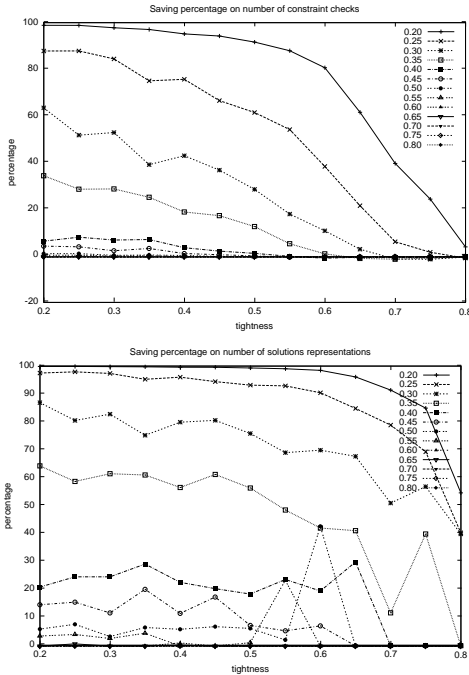
The overall transmutation process involves transmutating each variable domain, one by one, in some order. The order in which variables are processed affects the result. Figure (3.1),(3.2),(3.3) show sample transmutations: (3.1) is the original, (3.2) shows the result of processing in the order (V_2, V_1, V_3), while the CSP in (3.3) is obtained with either (V_1, V_3, V_2) or (V_3, V_1, V_2). (In fact, the CSP in (3.2) results from the transmutation of V_2 alone and subsequent attempts to transmutate V_1 and V_3 result in no change.)

The CSP in (3.2) has larger domains than the original CSP in (3.1), so it can be expected that it would take more time to solve. As a heuristic, we have chosen never to accept a domain transmutation if domain size is increased; we call this the *transmutation acceptance heuristic*. While variable domains can, in principle, be transmutated in any order, we always pick the variable whose transmutation leads to the maximum domain size reduction. To achieve this, we tentatively consider the transmutation of each variable in the network; if there is no variable whose domain size is reduced by the transmutation procedure, we terminate the process. The same approach is applied for subsequent variable transmutations. We need only recompute tentative transmutations for the domains of the variables that are immediate neighbors of the variable whose domain has been transmutated.

Experimental Results

We tested the `transmutate` algorithm and solved the transmutated problems for all solutions over randomly generated CSPs with ten variables[6] and six values in each domain, varying density from 0.2 to 0.8 with 0.05 increment step,

---

[6] These problems may appear small but, as will be discussed later, many real-world uses of the approach involve only small regions of larger network.

Saving percentage on number of constraint checks

Saving percentage on number of solutions representations

while tightness ranges from 0.2 to 0.8 with the same increment. We used model B random problem generator, with "flawless" constraint generation (Gent *et al.* 2001).

For each data-point, the number of constraint checks required to compute all solutions and the number of representations needed to express these solutions are computed and averaged over 100 problem instances. We gathered the data for both transmutated and original problems. The solver is MAC-3 with dom/deg variable ordering. For the transmutation cutoff heuristic, we fixed the upper bound of the `output` queue to be ten times the size of the domain (60).

It may appear that our experimental problems are too small, but remember that *all* solutions are to be computed. Since the solving time increases exponentially with the problem size, finding all solutions in larger networks therefore has few practical uses, and is not studied here. Indeed, as explained in the introduction, all solutions are needed only for relatively small subnetworks in larger CSPs — for creating meta-variables or for user perspicuity of subnetworks. Furthermore, since we tested the algorithm over a wide range of density and tightness, we had to settle on a relatively small problem size to keep the experimental time in check. However we believe the trend seen in the experiments in this paper can be extrapolated to larger problems without significant loss of accuracy.

Because transmutation time is always an insignificant fraction of the time needed to compute all the solutions of a CSP, it is always worthwhile trying the transmutation. In our setting, it takes just a few seconds to transmutate a $(50, 10, 0.5, 0.5)$[7] instance.

The results are shown in the graphs, in which the legends refer to density. While the transmutation cannot *guarantee*

---

[7]A tuple $(n, m, density, tightness)$ denotes a CSP where n is the number of variables and m is the domain size for each variable.

to reduce the number of constraint checks needed to compute all solutions to a CSP, it can produce huge savings. From the graphs, we see that more domain transmutation occurred for problems with lower density or lower tightness, as either case will lead to a lot of value combination; a similar reduction holds for the solution representation. On the other hand, in the worst case we have seen in the experimental data, which occurred only at a few data-points, the number of extra constraint checks for a transmutated CSP is no more than one or two percent of the untransmutated CSP.

Other Applications

Domain transmutation can be used for other purposes as well. Three applications are described in this section.

Robustness

We can exploit the combination of values after transmutation to achieve robustness for a single solution simply by ordering values during search based on the size of their labels. For example, suppose (a, abc, b, bc, c) is the transmutated domain of some variable. During search, it would be ordered as (abc, bc, a, b, c). The first solution obtained will be the most robust in this variable. That is, if a solution found involves abc, we know that it corresponds to three solutions in the original problem and should one of the three values be unavailable, it can be immediately replaced with either of the other two without any change to the rest of the solution. The advantage of this approach is that the transmutation process is independent of the solving process and thus no specialized solver is required. We can also order the variables for search based on which variables should be more robust to changes.

But, since variables are instantiated in order, there may be a case where the solver would prefer a solution which is very robust at one variable and brittle at another, rather than a uniformly robust solution: e.g., a solution tuple $(abcd, efg, hi, j)$ may be found before (ac, ef, hi, jk). (Ginsberg, Parkes, & Roy 1998) provides formalization and more details on solution robustness. In contrast, (Lesaint 1994) propose an algorithm that gives a single maximal bundling of solutions.

Decision Problems

In Definition 3, union is restricted to fragments that come from the same value so as to preserve the solutions. However, satisfiability remains unchanged when the restriction is lifted, although a solution cannot easily be converted back.

Definition 7 (Extended Union) Let a and b be two values in D_V. The *extended union* of a and b (denoted by $a \otimes b$) is a value c where $L_c = L_a \cup L_b$ and $\sigma_c(W) = \sigma_a(W) \cup \sigma_b(W)$ for all $W \in N_V$. The extended union is undefined (denoted by $a \otimes b = \emptyset$) if there exist two or more variables $X \in N_V$ such that $\sigma_a(X) \neq \sigma_b(X)$.

Theorem 6 (Satisfaction of Extended Union) Consider two values a and b in CSP $\mathcal{P}$ such that $a \otimes b \neq \emptyset$, and the resulting CSP $\mathcal{P}'$ where a and b are replaced by $a \otimes b$. $\mathcal{P}$ is satisfiable iff $\mathcal{P}'$ is satisfiable.

For example, consider $(\{a\}, \{x\}\times\{y\})$ and $(\{b\}, \{x\}\times\{z\})$ in D_V, where y and z are in D_W, $W \in N_V$. Both can be combined into $(\{a,b\}, \{x\}\times\{y,z\})$. Given a solution π involving $(\{a,b\}, \{x\}\times\{y,z\})$, there must be another solution involving either $(\{a\}, \{x\}\times\{y\})$ or $(\{b\}, \{x\}\times\{z\})$, although we cannot tell which one without inspecting $\pi(W)$.

In a situation where satisfiability of problems is the only concern, extended union can be used on its own to prune values during preprocessing or during search, similar to NI. And, unlike basic domain transmutation, the resulting domain is *guaranteed* to be smaller than the original.

Using extended union in domain transmutation can sometimes dramatically reduce the domain size. For example, the domain of V_2 in Figure (3.2) can be reduced from 13 values to 4 should we use extended union in the algorithm.

Finding a Single Solution

Even though Proposition 1 tells us that the upper bound on the search space for a transmutated CSP is smaller than its original, this is not in any way an indication of the actual search effort. In fact, in our preliminary, unreported experiments, the results are mixed. This is simply a consequence of conventional CSP solvers, where each value's support is established as the search progresses, and discarded afterward during backtracking. As the problem gets harder (more backtracking) and the domain size gets larger[8], finding a support for each value becomes the main bottleneck as it has to be re-discovered anew.

In this situation, the benefit of AC-4-like data structures (Mohr & Henderson 1986) in which supports for each values are remembered could offset its overhead. Although AC-4 was empirically shown to be inferior than AC-3 — a forgetful algorithm with non-optimal worst-case time-complexity — as far as we know there is no report on its effectiveness compared to other AC algorithms when used during search to solve *hard* problems. ((Sabin & Freuder 1994) compared MAC-4 with only forward checking.)

Related Work

Cross product representation (CPR) of partial solutions generated during search has been studied in (Hubbe & Freuder 1992). In extending a CPR to complete solutions, a new value is combined into existing CPR if possible; otherwise a new CPR is created. Domain transmutation could be seen as a "dynamic programming" version of the same concept.

Preliminary work on value transformation was reported in a poster paper by the authors of this submission [reference omitted for blind reviewing]. Subsequently and independently (Chmeiss & Saïs 2003) defined Generalized Neighborhood Substitutability, which corresponds exactly to the intersection of values. Rather than extracting identical parts from various values to form new ones, extra constraints are inserted to prevent the same search space from being explored again. However, this approach is undemonstrated, and no other benefits such as compact representation of solutions can be obtained.

---

[8]Larger domains in transmutated CSP are not uncommon, despite using the best variable ordering.

Conclusions

We present a new perspective on domain values in CSP where each value is viewed as a combination of smaller values. This allows us to freely manipulate variable domains so as to remove redundant partial assignments. We present an inexpensive method for merging duplicate local solutions in a constraint network, which subsumes NI and NS. The benefits are two-fold: reducing time spent on finding consistent assignments and reducing the complexity of enumerating the members of the complete solution set. Experimental results show that the savings are considerable, especially on loose problems with a low density of constraints. We use the new extended CSP definition, in which values are composed of labels and structures, to obtain new result on CDI and FI and to prove some properties of the algorithm presented. We also suggest how it could be used to find robust solutions, to solve decision problems, and to speed up the search for a single solution.

References

Benson, B. W., and Freuder, E. C. 1992. Interchangeability preprocessing can improve forward checking search. In *Proceedings of ECAI-92*, 28–30.

Chmeiss, A., and Saïs, L. 2003. About neighborhood substitutability in CSPs. In *CP-03 Workshop on Symmetry in Constraint Satisfaction Problems*.

Choueiry, B. Y., and Noubir, G. 1998. On the computation of local interchangeability in discrete constraint satisfaction problems. In *Proceedings of AAAI-98*, 326–333.

Freuder, E. C., and Hubbe, P. D. 1995. Extracting constraint satisfaction subproblems. In *Proceedings of IJCAI-95*, 548–555.

Freuder, E. C. 1991. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of AAAI-91*, 227–233.

Gent, I. P.; MacIntyre, E.; Prosser, P.; Smith, B. M.; and Walsh, T. 2001. Random constraint satisfaction: Flaws and structure. *Constraints* 6(4):345–372.

Ginsberg, M. L.; Parkes, A. J.; and Roy, A. 1998. Supermodels and robustness. In *Proceedings of AAAI-98*, 334–339.

Haselböck, A. 1993. Exploiting interchangeabilities in constraint satisfaction problems. In *Proceedings of IJCAI-93*, 282–287.

Hubbe, P. D., and Freuder, E. C. 1992. An efficient cross-product representation of the constraint satisfaction problem search space. In *Proceedings of AAAI-92*, 421–427.

Lesaint, D. 1994. Maximal sets of solutions for constraint satisfaction problems. In *Proceedings of ECAI-94*, 110–114.

Mohr, R., and Henderson, T. C. 1986. Arc and path consistency revisited. *Artificial Intelligence* 28(2):225–233.

O'Sullivan, B. 2002. Interactive constraint-aided conceptual design. *Journal of Artificial Intelligence for Engineering Design Analysis and Manufacturing (AIEDAM)* 16(4).

Sabin, D., and Freuder, E. C. 1994. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of ECAI-94*, 125–129.

Weigel, R., and Faltings, B. V. 1999. Compiling constraint satisfaction problems. *Artificial Intelligence* 115(2):257–287.

Weigel, R.; Faltings, B. V.; and Choueiry, B. Y. 1996. Context in discrete constraint satisfaction problems. In *Proceedings of ECAI-96*, 205–213.