Eliminating Redundancy in CSPs Through Merging and Subsumption of Domain Values

Chavalit Likitvivatanavong
School of Computing
National University of Singapore, Singapore
chavalit@comp.nus.edu.sg

Roland H. C. Yap
School of Computing
National University of Singapore, Singapore
ryap@comp.nus.edu.sg

ABSTRACT
Onto-substitutability has been shown to be intrinsic to how a domain value is considered redundant in Constraint Satisfaction Problems (CSPs). A value is onto-substitutable if any solution involving that value remains a solution when that value is replaced by some other value. We redefine onto-substitutability to accommodate binary relationships and study its implication. Joint interchangeability, an extension of onto-substitutability to its interchangeability counterpart, emerges as one of the results. We propose a new way of removing interchangeable values by constructing a new value as an intermediate step, as well as introduce virtual interchangeability, a local reasoning that leads to joint interchangeability and allows values to be merged together. Algorithms for removing onto-substitutable values are also proposed.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features—Constraints

General Terms
Algorithms, Theory

Keywords
Constraint Satisfaction Problems, Interchangeability, Onto-substitutability

1. INTRODUCTION
An important indicator of the hardness of a constraint satisfaction problem is the size of the search space. Eliminating interchangeable values was introduced in [7] as a way of reducing complexity of a problem by removing portions of the search space that are essentially identical. Recent focus on interchangeability has been on onto-substitutability [2, 6]: a domain value is onto-substitutable if any solution involving that value remains a solution when that value is replaced by some other value. Standard substitutability, by contrast, is a binary relation between two fixed elements.

In this paper, we redefine onto-substitutability as a binary relation and study its consequences. We propose joint interchangeability: two sets are joint-interchangeable iff any solution involving a value in one set remains a solution when that value is replaced by some value in the other set. Joint interchangeability is more practical since any one of the two sets can be eliminated; for onto-substitutability only a single value can be removed. We then propose virtual interchangeability: values are virtually interchangeable if they support the same values in every constraint but one. A set $S$ of virtually interchangeable values can be compactly represented by a value $s$, in effect making $S$ joint-interchangeable with $s$. Hence, virtual interchangeability leads indirectly to joint interchangeability.

To make sure virtually interchangeable set of values can be merged into a single value while retaining all the solutions, we expand the definition of domain value to accommodate the notion of label. A value may have more than one label, and labels are what actually appear as part of solutions. We introduce several new ways of comparing networks based on this concept. As a result, we can solve a CSP by transforming it using virtual interchangeability into a more compact network, solve the derived network, and convert it back—all without losing any solution. Moreover, this method works in the context of the hidden transformation, which is a way of transforming non-binary constraints into binary equivalents, by treating table constraints as a form of hidden variables. Preliminary results show that compressing all virtually interchangeable values is a promising approach to simplify table constraints in structured problems.

2. JOINT INTERCHANGEABILITY
Interchangeability and related ideas were first described in Freuder [7]. Bordeaux et al. [2] provided a formal framework that demonstrates connections among structural properties of CSPs. Removability, to which these properties reduce, is regarded by the authors to be the basis of how a value can be removed without affecting satisfiability of the problem. Freuder [6] later proposed the concept of dispensability: a value is dispensable if removing it will not remove all solutions to the problem. A value can then be dispensable without being removable (or onto-substitutable as called in [6]). Dispensability would therefore appear to be a more fundamental property than onto-substitutability. A survey of interchangeability concepts is reported in [9].
Onto-substitutability underscores the notion that a value’s attributes can be broken down and subsumed by other values. An onto-substitutable value can be removed without affecting satisfiability of the problem precisely because it is semantically redundant. For this reason, onto-substitutability is arguably key to understand many interchangeability concepts, but not as fundamental as dispensability when it comes to determine why a value can be removed.

Given the prospects of onto-substitutability, we will focus on this property and its derivations. First, we give the formal definition of CSP and redefine substitutability so as to make onto-substitutability a binary relation as follows. Any set of values mentioned in this section must be nonempty.

**Definition 1 (CSP).** A finite constraint network $P$ is a triple $(X, D, C)$ where $X$ is a finite set of $n$ variables and $C$ a finite set of $e$ constraints. $D(X) \in D$ represents the set of values allowed for variable $X \in X$. Each constraint $C \in C$ involves an ordered subset of variables in $X$ called scope (denoted by scp$(C)$), and an associated relation (denoted by rel$(C)$). For any k-ary constraint $C$ with scp$(C) = \{X_1, \ldots, X_k\}$, rel$(C) \subseteq \prod_{i=1}^{k} D(X_i)$. For any k-tuple $t = (a_1, \ldots, a_k)$ over $T = \{X_1, \ldots, X_k\}$ such that $X_i \in X$ and $a_i \in D(X_i)$, $t[X_i]$ denotes $a_i$, and scp$(t)$ denotes $T$. If $S \subseteq T$ then $t[S]$ denotes the tuple over $S$ obtained by restricting $t$ to the variables in $S$. A solution of $P$ is a member of rel$(P) = \{ t \mid scp(t) = X \land \forall C \in C. t[scp(C)] \in rel(C) \}$. $P$ is satisfiable iff rel$(P) \neq \emptyset$.

We present new definitions and re-define substitutability and onto-substitutability as follows.

**Definition 2.** Given value $v \in D(X)$, the maximally substitutable set of $v$ is maxsub$(v) = \{ b \in D(X) \mid$ there exists a solution involving $v$ which remains a solution when $v$ is substituted by $b \}$.

**Definition 3.** A value $v \in D(X)$ is substitutable by a set of values $S$ iff $S \subseteq$ maxsub$(v)$ and any solution involving $v$ remains a solution when $v$ is substituted by some $b \in S$. $S$ is called a substitutable set of $v$.

We simply say $v$ is substitutable if there exists $S$ such that $v$ is substitutable by $S$. When $|S| > 1$, $v$ is onto-substitutable and that $S$ is an onto-substitutable set of $v$. A substitutable set of $v$ is minimal if there is no strictly smaller substitutable set of $v$. Distinct minimally substitutable sets may exist for any given value. We also extend substitutability so that a set of value $S$ is substitutable by $T$ iff for any value $v \in S$, $v$ is substitutable by $T$.

Interchangeability can be redefined to cover many-to-many relationship in a similar fashion.

**Definition 4.** A set of values $S \subseteq D(X)$ is said to be joint-interchangeable (JI) with a set of values $T \subseteq D(X)$ iff $S$ is substitutable by $T$ and $T$ is substitutable by $S$.

When $|S| = 1 = |T|$, JI reduces to ordinary interchangeability. JI is symmetric and transitive. $S$ is minimally joint-interchangeable with $T$ if there exist no $S' \subseteq S$ and $T' \subseteq T$ such that $S'$ is JI with $T'$ and $S' \neq S \subseteq T' \neq T$.

**Example 1.** Consider a network involving two variables with solutions $\{(1, a), (2, a), (3, b), (4, b), (5, c), (6, c), (1, d), (3, d), (5, d), (2, c), (4, c), (6, e)\}$. $\{a, b, c\}$ is minimally joint-interchangeable with $\{d, e\}$.

Take note that the definitions of substitutability and JI allow the same value to appear on both sides of the relations. This helps us identify JI sets that would not otherwise be recognized. The following example illustrates.

**Example 2.** Consider a set of solutions $\{(1, a), (2, a), (1, b), (2, c), (3, b), (3, c)\}$. $a$ is not interchangeable with $b$, but $\{a, c\}$ is minimally joint-interchangeable with $\{b, c\}$.

**Proposition.** If $S$ is joint-interchangeable with $T$, then either $S \setminus T$ or $T \setminus S$ can be eliminated without affecting the network’s satisfiability.

In [2] the authors claim that local reasoning is not sound for onto-substitutability. Freuder [6] shows that this is not the case if the scope of “local reasoning” is broadened to include closure on sub-problems. The same rationale can be applied to joint interchangeability.

JI is a stronger than onto-substitutability but proves to be more useful. The latter lets us eliminate objects only from one side of the relation, whereas JI allows either side to be removed. Since a CSP with more values generally translates to longer search, once it is known $S$ is JI with $T$ an easy way to simplify the problem is to eliminate the larger set. Conversely, we want to identify two JI sets such that their size difference is as large as possible. We introduce and study a local reasoning which takes advantage of this fact called virtual interchangeability in §4.

Onto-substitutability can be weakened further by considering substitutability in only some solution.

**Definition 5.** A value $v$ is nominally substitutable by $b$ iff there exists a solution involving $v$ and it remains a solution when $v$ is substituted by $b$. A value $v$ is simply said to be nominally substitutable when there exists such $b$, that is, when maxsub$(v) \neq \emptyset$.

Nominal substitutability and dispensability depend on the existence of a solution. Local reasoning such as closure is thus ineffective, because extending some solution in a bounded area to the whole problem is just as computationally difficult as finding a new solution from scratch. Nominal substitutability is equivalent to “minimal substitutability” in [6] and “context dependent interchangeability” in [16].

Recombination of values in the smallest closure was studied in [3] and it was shown that two values are nominally substitutable iff their structure in the closure overlap and a fragment of their intersection involved is in a solution.
3. ENHANCING DOMAIN VALUES

Domain values serve two main purposes: structurally and semantically. These two aspects are intertwined in most CSP models, yet we would have more flexibility in manipulating a network when they are decoupled. To this end, we extend the definition of a value in this section. Combined with virtual interchangeability, this allows us to merge values while preserving all solutions at the same time.

Definition 6 (Labels). A value \( v \) of a constraint network \( \mathcal{P} \) is a tuple \((\text{id}, \text{lab})\) where \( \text{id} \) is an identifier unique to this value in \( \mathcal{P} \) (also denoted by \( \text{uid}(v) \)) and \( \text{lab} \) is a set, which we also call the value’s labels. (also denoted by \( \text{lab}(v) \)). A value must have at least one label. Choosing a label for \( v \) from \( \text{lab}(v) \) is called an interpretation of \( v \).

The enhanced definition permits a value to be associated with multiple labels. Different values are allowed to share the same labels.

We consider two possible evaluations of labels when a value is part of a network’s solution. In the first approach, every label is legitimate in any solution. Alternatively, only one of the labels is guaranteed to be legitimate. This leads to two new ways for comparing networks involving values with multiple labels: equivalence and conformity.

Definition 7. Given tuple \( t \) where \( \text{scp}(t) = \{X_1, \ldots, X_k\} \), \( \text{sol}(t) \) denotes \( \prod_{i=1}^{k} \text{lab}(t[X_i]) \). Given networks \( \mathcal{P} \) and \( \mathcal{Q} \),

- \( \text{sol}(\mathcal{P}) \) denotes \( \bigcup_{t \in \text{sol}(\mathcal{P})} \text{sol}(t) \), a member of which is called a rendered solution of \( \mathcal{P} \).
- \( \mathcal{P} \) and \( \mathcal{Q} \) are equivalent iff \( \text{sol}(\mathcal{P}) = \text{sol}(\mathcal{Q}) \).
- \( \mathcal{P} \) subsumes \( \mathcal{Q} \) iff \( \text{sol}(\mathcal{P}) \supseteq \text{sol}(\mathcal{Q}) \).
- for any \( t \in \text{rel}(\mathcal{P}) \), if \( \exists s \in \text{sol}(t) \) such that \( s \in \text{sol}(\mathcal{Q}) \) then we say the interpretation of \( t \) is sound in \( \mathcal{Q} \), and that \( s \) is an interpretation of \( t \) in \( \mathcal{Q} \).
- \( \mathcal{Q} \) conforms to \( \mathcal{P} \) iff for any \( t \in \text{rel}(\mathcal{Q}) \) there exists a sound interpretation of \( t \) in \( \mathcal{P} \).

An example of equivalent networks is shown in Figure 1.

Theorem 1. Given CSPs \( \mathcal{P} \) and \( \mathcal{Q} \), the following holds,

- if \( \mathcal{P} \) and \( \mathcal{Q} \) are equivalent then \( \mathcal{P} \) conforms to \( \mathcal{Q} \) and \( \mathcal{Q} \) conforms to \( \mathcal{P} \). The converse does not hold. That is, if \( \mathcal{P} \) conforms to \( \mathcal{Q} \) and \( \mathcal{Q} \) conforms to \( \mathcal{P} \), it is not necessary that \( \mathcal{P} \) and \( \mathcal{Q} \) are equivalent.
- if \( \mathcal{P} \) conforms to \( \mathcal{Q} \) and \( \mathcal{Q} \) conforms to \( \mathcal{P} \), then \( \mathcal{P} \) is satisfiable iff \( \mathcal{Q} \) is satisfiable.

4. VIRTUAL INTERCHANGEABILITY

Identification followed by elimination has been an established method for dealing with redundant values in CSPs. This practice is simple and straightforward. As a result, much of the attention has been devoted to finding a new kind of interchangeability and developing more efficient algorithms that recognize these properties.

We present a proactive strategy with regard to properties that are binary relations. Given a relation \( R \), rather than the usual passive approach of finding two objects \( x \) and \( y \) such that \( xRy \) and eliminating one of the two, we need only identify \( x \), then create \( y \) from \( x \) so that \( xRy \), and finally eliminate \( x \). Central to this approach, however, is whether a “better” object \( y \) can be created from \( x \). Concretely for JI, instead of identifying two sets that are JI with each other, we focus on finding a set of values that possess a certain property, then by exploiting that property create an equivalent value that represents that set more concisely, and only after the new value is added do we remove the original values.

In this section we introduce the concept of virtual interchangeability, a local reasoning that can be used as outlined above. We begin by recalling necessary definitions.

Definition 8. The projection of constraint \( C \) to \( S \subseteq \text{scp}(C) \)
is a constraint $\pi_S(C)$ where $\text{scp}(\pi_S(C)) = S$ and $\text{rel}(\pi_S(C)) = \{t[S] \mid t \in \text{rel}(C)\}$. The projection of a tuple is defined in the same fashion. The concatenation of $t_1 \in C_1$ and $t_2 \in C_2$ ($\text{con}(t_1, t_2)$) is the tuple $t$ resulting from the concatenation of $t_1$ and $t_2$ followed by rearrangement so that $\text{scp}(t) = \text{scp}(t_1) \cup \text{scp}(t_2)$.

**Definition 9.** Given two values $a, b \in D(X)$ and constraint $C$ such that $X \in \text{scp}(C)$, values $a$ and $b$ are neighborhood interchangeable with respect to $C$ if and only if

$$\{t \in \bar{D} \mid \text{con}(a, t) \in \text{rel}(C)\} = \{t \in \bar{D} \mid \text{con}(b, t) \in \text{rel}(C)\}$$

where $\bar{D} = \pi_{\text{scp}(C)} \setminus (X)\{C\}$.

**Definition 10.** Two values $a, b \in D(X)$ are neighborhood interchangeable (NI) $[7, 13]$ if and only if they are neighborhood interchangeable with respect to $C$ for every constraint $C$ such that $X \in \text{scp}(C)$.

Given NI values, we can combine them into a single value without losing any solution simply by merging their labels into those of the representative value while discarding the remaining values. As a result, the initial network and the network after the NI values are merged are equivalent.

**Definition 11.** Two values $a, b \in D(X)$ are virtually interchangeable (VI) (with respect to $C$) if there is at most one constraint $C$ such that $X \in \text{scp}(C)$ and $a$ and $b$ are not neighborhood interchangeable with respect to $C$. A set of values are virtually interchangeable if any two values are virtually interchangeable with each other.

VI and NI are almost the same except for the difference of supports in a single constraint. NI implies VI but VI does not imply NI. Neighborhood Substitutability (NS) $[7]$ is incomparable to VI. A network that contains no VI values is called VI-free.

**Theorem 3.** Given $a, b \in D(X)$ in $P$, a new network $Q$ can be derived from $P$ by merging $b$ into $a$ as follows

1. updating $\text{rel}(C)$ for any $C$ involving $X$ by altering any tuple $t \in \text{rel}(C)$ where $t[X] = b$ so that $t[X] = a$
2. setting $\text{lab}(a) \cup \text{lab}(b)$ to be the new value of $\text{lab}(a)$
3. removing $b$ from $D(X)$

Consequently,

- $Q$ subsumes $P$
- $P$ conforms to $Q$ and $Q$ conforms to $P$

And as a result of Theorem 1 and 2, it follows that,

- $P$ is satisfiable iff $Q$ is satisfiable.
- For any rendered solution $s$ in $P$ there exists a solution $t$ in $Q$ such that $s$ is an interpretation of $t$ in $P$.

Theorem 3 tells us there always exists a sound interpretation of $t$ in $P$ for any $t \in \text{rel}(Q)$. That is, the first solution found in $Q$ will always lead to a rendered solution in $P$, pending the interpretation. Interpreting a solution must be done only in the original network to maintain soundness and avoid spurious solutions.

**Theorem 4.** Assume the condition in Theorem 3 and in addition suppose that value $a$ is virtually interchangeable with value $b$, after merging $b$ into $a$ the network $Q$ is equivalent to $P$.

Theorem 3 and 4 show that VI values can be combined into a new value so that the original values and the new value are joint-interchangeable. A new network obtained by adding that new value and removing the original set will be equivalent to the original network. Because both networks are equivalent, the interpretation is search-free. The theorem can be extended to a group of VI values.

**Example 4.** Consider $D(X)$ of the network in Figure 3 (top). Value labeled 0 is VI with value labeled 1, while value labeled 2 is VI with value labeled 3. The network on the bottom is derived according to Theorem 3. Solutions of the top are preserved in the bottom network, which include some spurious rendered solutions. For instance, consider the solution $t = (2, 0, 1, 0)$ of the bottom network. An interpretation of $t$ in this network itself would produce $2, 0, 0$, a rendered solution that does not belong to the original. However a sound interpretation of $t$ in the top network exists as a result of Theorem 4. We pick a correct label of $t[X]$ by inspecting the top network and find out which value in $D(X)$ supports $t[Y]$ and $t[Z]$. In this case, label 1 is the answer.

Given a network, we can derive a more compact one by repeatedly applying Theorem 3. This involves going through each variable one by one and merging all VI values where possible. Results are propagated to the adjacent variables. The process terminates when no more VI values are detected. Propagation is necessary because values that are not initially VI may become so, once their neighbors are modified. Consider Example 3 for instance. In $P'$ each domain has only one value, which is the best compression possible. Initially however, only the two variables at both ends ($X_1$ and $X_n$) can be compressed. No VI values are detected in the middle variables. After $X_1$ and $X_n$ are compressed, the adjacent variables $X_2$ and $X_{n-1}$ have to be re-examined in light of the change. Values of $D(X_2)$ and $D(X_{n-1})$ then all become VI and will be merged as a result. The process continues until the propagation converges.

It is important to note that merging all VI values of a variable domain with respect to different constraints produces different results. Because a variable may contain different VI values that are VI with respect to different constraints, to reduce the size of the search space one heuristic is to pick some ordering so that the domain size becomes as small as possible when no VI values can no longer be found. In this paper, we consider only this greedy heuristic. Given a domain, we detect VI values by considering each involved constraint one by one and calculate the possible reduction in domain size. The constraint that gives the best reduction is chosen first for the application of Theorem 3. The reduction for each constraint is re-computed and the process is repeated until all VI values for this variable are merged.

In many cases, as Example 4 has shown, a sound interpretation can be deduced by looking at the instantiation of the neighboring variables in the original network, provided that they have singleton labels. But in general interpretation is not search-free, as shown in Example 3. When a solution
$t \in \text{rel}(C)$ involves two values each having multiple labels, it is not simple to have a fast and easy way to determine a sound interpretation. However, the cost can be reduced to just a matter of simple look-up by sacrificing some compressibility. This is achieved by ensuring that no values with multiple labels can support each other by imposing the following restriction on Theorem 3 before $b$ is merged into $a$:

**Proposition 2.** Consider the case for any constraint $C$ such that $X \in \text{scp}(C)$, where there is not any $t \in \text{rel}(C)$ such that $t[X]$ is either $a$ or $b$ and $|\text{lab}(t[X])| > 1$ for some $X' \neq X$. In such cases, no propagation is necessary.

The broken-triangle property (BTP) [5] has a similar condition to VI in that it also forbids values having different sets of supports in two adjacent variables. Figure 4 shows that the BTP is not comparable to the VI-free property.

### 5. MERGING VI VALUES

We now present algorithms for finding and merging VI values of a single variable domain. Given a variable, all involving constraints are initially joined together into a single table constraint. The result is then compressed by considering each constraint in turn whether values are VI with regard to this constraint. We consider only variables involving binary constraints. Later, we describe how to compress non-binary constraints.

Given variable $X$, Algorithm 1 collects all supports of values in $D(X)$ and tabulates the results. Each row represents supports of a single value in $D(X)$ from various constraints, and each cell contains the supports’ labels, in effect making the row, a cartesian product representation (CPR). The table can be viewed as partially compressed from the start.

Algorithm 2 details the greedy compression process. To compress a table, we hypothetically evaluate whether values are VI with respect to each column. The actual compression is performed with regard to the column that yields the best reduction (stopping at this point is denoted as the single-best compression process). The process is repeated until every column is committed. Constraints are updated using the finished table as the final step.

**Proposition 3.** After all VI values are merged with respect to each constraint, any further merging of VI values in this domain with respect to the same constraint is not possible so long as neighboring domains are not altered.

**Proof.** At time $t_1$ all VI values are merged with respect to constraint $C$, no two values are NI with respect to all constraints except $C$. If at some later time $t_2$, there exist two values that are NI in all but $C$, these two values must differ in at least one other constraint $C'$ beside $C$. Consequently, the difference in $C'$ must be eliminated at some point between $t_1$ and $t_2$. The change in $C'$ affects the neighboring domains, contradicting the assumption.

**Theorem 5.** The cost of $\text{Compress}(C)$ is $O(mn \log n)$, where $C$ is a table constraint of $m$ columns and $n$ rows.

**Proof.** Line 5 takes $O(n)$. It takes $O(mn \log n)$ in line 6 to sort the projection from line 5. After sorting, counting the duplicate in line 7 is just a matter of going through each member of $A$ while comparing and the previous member, costing $O(mn)$. The whole process then takes $O(\sum_{i=1}^{m} \text{inn \log n}) = O(mn \log n)$.

**Example 5.** We consider merging VI values of $X$ in the network in Figure 3 (top). After supports are collected, the table is shown in Table 1(a). Tentative reduction size with respect to $C_{XY}$ and $C_{XZ}$ are both 2. The table is then compressed by merging VI values with respect to $C_{XY}$. The result is shown in Table 1(b). Next, VI values with respect to $C_{XZ}$ are combined, resulting in Table 1(c). No more VI values exist and this table will be used to update $C_{XY}$ and $C_{XZ}$, as depicted in Figure 3 (bottom). There are two values in the new $D(X)$: one with labels $\{0, 1\}$, and the other with $\{2, 3\}$. Alternatively, one can choose to expand Table 1(a) first so that it would contain 9 tuples, each component of each tuple
Table 1. Joined tables for constraints involving $X$.

<table>
<thead>
<tr>
<th>$X$</th>
<th>$Y$</th>
<th>$Z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0,1}</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>{0,1,2}</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>(0,1)</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>(0,2)</td>
</tr>
</tbody>
</table>

Table 2. Merging of a 4-ary constraint. Labels in the $H$ column correspond to domain values of the hidden variable.

<table>
<thead>
<tr>
<th>$H$</th>
<th>$X$</th>
<th>$Y$</th>
<th>$Z$</th>
<th>$W$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$b$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$c$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$d$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$e$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$f$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$g$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3. Reduced tables for constraints involving $X$.

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>$Y$</td>
</tr>
<tr>
<td>$a$</td>
<td>0</td>
</tr>
<tr>
<td>$b$</td>
<td>0</td>
</tr>
<tr>
<td>$c$</td>
<td>0</td>
</tr>
<tr>
<td>$d$</td>
<td>1</td>
</tr>
</tbody>
</table>

Proposition 4. Given a non-binary constraint network $\mathcal{P}$ where each value has a single label initially. Assume the network is converted into a binary network using the hidden transformation. Assume further that we merge all possible VI values in every variable domain, including dual and ordinary variables, and propagate as necessary, resulting in network $\mathcal{Q}$. Given $t \in \text{rel}(\mathcal{Q})$ deciding whether interpretation of $t$ in $\mathcal{P}$ is sound is search-free.

Proof. There is no need for the interpretation of a dual variable because it does not exist in the original network. Instantiation of the dual variables determines the interpretation of the ordinary ones. Note that this reasoning is not valid for a mixed binary and non-binary network since there would be no intermediary dual variable between two original variables if there exists a binary constraint involving them.

Example 6. Consider the constraint in Table 2(a). Let us consider merging VI values of $H$, the hidden variable for this constraint. Tentative reduction size with respect to each constraint are: $C_{HX} = 3$, $C_{HY} = 2$, $C_{HZ} = 1$, $C_{HW} = 0$. Since the maximum reduction size is 3, we then proceed with merging VI values with respect to $C_{HX}$. Table 2(b) shows the result. At this point, we recompute the reduction size for each remaining constraint: $C_{HY} = 1$, $C_{HZ} = 0$, $C_{HW} = 0$. Table 2(c) shows the result after further merging with respect to $C_{HY}$. No more compression is possible. Since instantiation of hidden variables has no effect on actual solution, there is no need to have multiple labels for values in $H$. We can re-label aceg, bf and d to 1, 2, and 3, for instance. Table 2(c) will be used to construct the hidden constraints $C_{HX}$, $C_{HY}$, $C_{HZ}$, and $C_{HW}$.

We note that although one can hypothetically view a table as a dual variable, but without actual dual variables present, the structural information of the combined tuples is lost. The CPR thus represents both syntax and semantic of the constraints at the same time. Consistency algorithms that normally work on tuples therefore must be modified to handle the CPR. On the other hand, the network that is explicitly transformed by the hidden transformation and compressed afterwards does not require any specialized algorithm.

5.1 Experimental Results

We now present some preliminary results on domain and table compression. We first compare results on random CSPs, which we expect to be unstructured, followed by structured problems. Results for series generated according to model RB are shown in Table 3. The processing time for each instance is negligible — a small fraction of a second at the most. Enumerating each table before the compression yields almost no improvement. Despite the fact that only a small percentage of all values in a single instance are found to be VI, we do not find even a single pair of VI values in any instance tested. Interestingly, the number of affected variables is significant though the reduction is small, i.e. up to 27% of variables but with only 1.63% values merged. Since detecting VI values dynamically [8, 11] can improve their numbers, we can expect even more improvement for VI.

NS is not considered here because finding all NS values is too expensive and only two values can be checked at the same time. The number of pairs grows exponentially along with domain size. By contrast, algorithms for NI and VI can tackle the whole domain at once in polynomial time.

Table 5 displays the results for crossword puzzle $\text{ukVG}^2$, which involves non-binary constraints of non-random structure: a constraint of arity $k$ contains all the words of length $k$. In contrast to the randomly generated binary problems, we see that compressing all VI values yields remarkable reduction.

rates, as high as 70% for the arity-4 constraint. At low arity, there is not much difference between the single-best compression and the greedy compression, but as the arity increases the reduction rate for the single-best drops sharply while the greedy compression maintains its rate well. The greedy compression can merge twice more values than the single-best compression at high arity.

We have also conducted some experiments to measure the running time of compressed instances from the series rand-3-20-20 and rand-5-30-5 using Abscon\(^3\) as a black box solver. Because newer solvers such as Abscon have implemented advanced GAC algorithms, we expect them to perform well on table constraints and relatively poorly on the hidden-transformed binary encoding, which requires more variables, more constraints, and much larger domains. We therefore compare the running time of the original problems against their hidden transformation problems that are simplified by the compression. The reduction rate ranges from 60% to 86%. Overall, the running time of the transformed problems is very competitive with that of the original — winning over half of all the instances from rand-3-20-20 and losing slightly on rand-5-30-5 for most instances — despite the obvious disadvantages of the hidden transformation. These results however are limited by the fact that the solver is not informed of the arity, the tightness of each constraint, and the percentage of values merged via VI, and the percentage of variable affected (\(\geq 2\) VI values merged.)

### 6. IDENTIFYING REDUNDANT VALUES

We turn our attention to onto-substitutability in this section. An onto-substitutable value is redundant in the sense that every solution it participates in is also covered by some other value. We consider only algorithms that compute all onto-substitutable values in the smallest closure of a given variable in a network with extensional constraints.

To remove all onto-substitutable values from the domain of variable \(X\), we do the following:

1. let \(D\) be the table created as a result of joining of all table constraints \(C\) involving \(X\).
2. sort \(D\) in lexicographic order while ignoring column \(X\).

3. merge cells of any two rows that differ only at col. \(X\).
4. while there exists a value \(v \in D(X)\) such that every cell containing \(v\) also contains another value \(v'\), remove \(v\) from these cells and from \(D(X)\).

An example is demonstrated in Table 4. Variable \(A\) is involved in two ternary constraints given as positive tables \((a)\) and \((b)\). After the two tables are joined and rows are merged, the result is shown in \((c)\). Let us first consider value 0. In column \(A\), 0 is never contained in a singleton set. Therefore 0 is onto-substitutable and we can remove it from the \(D(A)\) and from all the sets in column \(A\). Value 1 does not appear as a singleton as well; hence 1 is onto-substitutable. The set containing value 2 in the first row has become a singleton after the removal of 0, so 2 is not onto-substitutable. So are values 3 and 4 for the same reason. Therefore, values 0, 1, 3, and 4 are onto-substitutable and can be removed from \(D(A)\).

Note that the concept of onto-substitutability is tied to the structure of the network, therefore algorithms may produce different outcomes depending on the sequence of value removal. For instance, suppose value 5 is considered first, rather than value 0 as done in the previous example. Value 5 is onto-substitutable and it is removed. Next, values 2, 3, and 4 are examined and found to be onto-substitutable and removed as well. As a result, values 5, 2, 3, and 4 are found to be onto-substitutable in that order.

While this algorithm is simple and straightforward it has serious inefficiency in the process of joining of the constraints involved, which amounts to joining every constraint in a complete-graph constraint network for instance. We will consider an improved algorithm which places more computation on each table and operates on joined sub-tables only when necessary. Details are given in Algorithm 3.

We assume tables are initially merged as done in Table 4 (d) and (e) (from (a) and (b)). The algorithm decides whether \(a \in D(X)\) is onto-substitutable. It first checks if there exist values that do not appear in all tables (line 3). These values cannot be joined and are filtered out. This step is equivalent to enforcing maxRPWC \([12]\). The next step tests whether

<table>
<thead>
<tr>
<th>Instance</th>
<th>#v</th>
<th>#d</th>
<th>#c</th>
<th>#t</th>
<th>#vi</th>
<th>#va</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-10-5-15-658</td>
<td>10</td>
<td>5</td>
<td>15</td>
<td>0.658</td>
<td>2.58%</td>
<td>10.90%</td>
</tr>
<tr>
<td>2-20-3-30-519</td>
<td>20</td>
<td>3</td>
<td>30</td>
<td>0.519</td>
<td>4.38%</td>
<td>12.35%</td>
</tr>
<tr>
<td>2-20-20-25-909</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>0.909</td>
<td>1.58%</td>
<td>18.60%</td>
</tr>
<tr>
<td>2-50-3-120-367</td>
<td>20</td>
<td>3</td>
<td>120</td>
<td>0.367</td>
<td>2.54%</td>
<td>6.96%</td>
</tr>
<tr>
<td>2-50-5-70-683</td>
<td>50</td>
<td>5</td>
<td>70</td>
<td>0.683</td>
<td>2.59%</td>
<td>11.08%</td>
</tr>
<tr>
<td>2-100-4-200-500</td>
<td>100</td>
<td>4</td>
<td>200</td>
<td>0.500</td>
<td>2.05%</td>
<td>7.36%</td>
</tr>
<tr>
<td>2-100-10-110-877</td>
<td>100</td>
<td>10</td>
<td>110</td>
<td>0.877</td>
<td>1.95%</td>
<td>14.96%</td>
</tr>
<tr>
<td>2-200-100-220-985</td>
<td>200</td>
<td>100</td>
<td>220</td>
<td>0.985</td>
<td>1.63%</td>
<td>27.20%</td>
</tr>
</tbody>
</table>

Table 3. Results for random problems. The results for each series are averaged over 100 instances. \#v is the number of variables in each instance, \#d the domain size, \#c the number of constraints, \#t the tightness of each constraint, \#vi the percentage of values merged via VI, and \#va is the percentage of variable affected (\(\geq 2\) VI values merged.)

<table>
<thead>
<tr>
<th>Instance</th>
<th>#v</th>
<th>#d</th>
<th>#c</th>
<th>#t</th>
<th>#vi</th>
<th>#va</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-100-4-200-500</td>
<td>100</td>
<td>4</td>
<td>200</td>
<td>0.500</td>
<td>2.05%</td>
<td>7.36%</td>
</tr>
<tr>
<td>2-100-10-110-877</td>
<td>100</td>
<td>10</td>
<td>110</td>
<td>0.877</td>
<td>1.95%</td>
<td>14.96%</td>
</tr>
<tr>
<td>2-200-100-220-985</td>
<td>200</td>
<td>100</td>
<td>220</td>
<td>0.985</td>
<td>1.63%</td>
<td>27.20%</td>
</tr>
</tbody>
</table>

Table 4. Tables for constraints involving \(A\).
it is possible for \( a \) to be subsumed. Values that appear as singletons can never be onto-substitutable (line 3). This step has been employed in the basic algorithm. The algorithm may take this step first but it has to do it again after the maxRPWC filtering because values that are not singleton initially may become so later.

The algorithm then calculates the number of tuples involving value \( a \) (goal). This can be done by simple multiplication without actually joining the tuples across tables. The upper bound (ub) is the maximum number of tuples that are shared by other values. Again, this can be easily computed and if the number is lower then the value \( a \) can never be subsumed and the algorithm can terminate and give a negative answer. Otherwise, the algorithm will try to enumerate all the tuples that share \( a \)’s structure and see whether \( a \) can be completely covered (line 3) via the set store. Alternatively, this stage can be implemented more compactly using tuple sequence [14] marked by lower and upper bounds, instead of storing the actual tuples themselves.

Example 7. Consider determining whether \( 0 \in D(A) \) from Table 4 (a) and (b) is onto-substitutable. The algorithm will check these four sets \{0, 2, 3\}, \{0, 1, 4, 5\}, \{0, 2, 4, 5\}, and \{0, 2, 3, 5\} (the tuple involving \{1\} has nothing to do with the onto-substitutability of \( 0 \)). First, value 1 is removed from \{0, 1, 4, 5\} because there would be no tuple involving 1 after joining the two tables. There is no singleton containing 0 so the algorithm passes through the cutoff test (line 3). Next, goal is calculated to be 4, while ub is the size of the CPR involving value 2 (2) + value 3’s (1) + value 4’s (1) + value 5’s (2) = 6. The algorithm also passes through the cutoff test (should the original two tables contain no tuple involving 2 and 5 the values of ub would be 2, for instance, and false would be returned in this case). The algorithm then adds the following tuples sequentially to store: \{(0, 0, 2, 2), (0, 0, 3, 3), (1, 1, 2, 2), (1, 1, 3, 3)\}. At this point store’s size is equal to goal so \( a \) is proved to be onto-substitutable.

Algorithm 3: is-onto-substitutable\((X, a)\)

1. pick some constraint \( C' \) such that \( X \in scp(C') \);
2. foreach tuple \( t \in rel(C') \) such that \( a \in t[X] \)
   
   1. foreach \( b \in t[X] \) such that \( b \neq a \)
   
   1. if \( b \notin t'[X] \) in each \( C' \neq C' \) for some \( t' \)
      
      remove \( b \) from \( t[X] \);
   
   3. foreach \( C \) such that \( X \in scp(C) \)
      
      1. if \( |t[X]| = 1 \) for some \( t \in rel(C) \)
         
         return false;
   
   4. goal \( \leftarrow \sum_{t}[t], a \notin t[X] \) and \( t \) is joined across all constraints;
   
   5. \( ub \leftarrow \sum_{t}[t], a \notin t[X] \) and \( t \) is joined across all constraints;
   
   6. if \( ub < goal \) then return false;
   
   7. store \( \leftarrow \emptyset \);
3. foreach tuple \( t \in rel(C) \) such that \( a \in t[X] \)
4. foreach \( b \in t[X] \) such that \( b \neq a \)
5. \( u \leftarrow \text{CPR} \) joined across all constraints s.t. \( b \in u[X] \);
6. foreach \( t' \) enumerated from \( u \)
7. store \( \leftarrow \text{store} \cup \pi_{scp(C)\backslash\{X\}}(t) \);
8. if \( |\text{store}| = \text{goal} \) then
9. return true;

return false;

7. CONCLUSIONS

Onto-substitutability has been shown as a sufficient condition for a value to be treated as redundant. We extend its definition from unary to binary relation and introduce JI as a replacement for interchangeability. As a symmetric binary relation, JI allows us to remove either one of the interchangeable set of values, giving us more flexibility as a result. We then present an original strategy for dealing with redundant values: detection, creation, followed by elimination. Since JI is a binary relation, this strategy makes sense: we can identify a group of values, create a JI-equivalent using fewer number of values, and then delete the original values.

The definition of values is expanded to include the concept of labels, which allows us to tease out semantics from the structure of the network. Different CSPs can be compared solely on their semantics (via their rendered solution). We then introduce VI, a new local reasoning that leads to JI. While it remains to be seen whether future work in this area will give us a new local property that also leads to JI, we have empirically shown the promise of VI as a compression tool. Table constraints can be compressed using other techniques such as decision trees [10], but they require specialized consistency algorithms unlike VI. VI may prove useful in different situations as well, for instance, VI could be used in addition to NS as a simplification operation before reasoning with CSP patterns [4].

In [2], the authors raise “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?” We believe VI is one such property, provided that creation of new values is permitted. Allowing networks to be augmented in this manner could also lead to a more powerful general framework.

8. ACKNOWLEDGMENTS

This work has been supported by the Ministry of Education under Grant No. MOE2012-T2-1-155.

9. REFERENCES


