Neighborhood Interchangeability and Dynamic Bundling for Non-Binary Finite CSPs

Anagh Lal and **Berthe Y. Choueiry**
Constraint Systems Laboratory
University of Nebraska-Lincoln
{alal | choueiry}@cse.unl.edu

Eugene C. Freuder
Cork Constraint Computation Center
University College Cork
e.freuder@cs.ucc.ie

Abstract

Neighborhood Interchangeability (NI) identifies the equivalent values in the domain of a variable of a Constraint Satisfaction Problem (CSP) by considering only the constraints that directly apply to the variable. Freuder described an algorithm for efficiently computing NI values in binary CSPs. In this paper, we show that the generalization of this algorithm to non-binary CSPs is not straightforward, and introduce an efficient algorithm for computing NI values in the presence of non-binary constraints. Further, we show how to interleave this mechanism with search for solving CSPs, thus yielding a *dynamic bundling* strategy. While the goal of dynamic bundling is to produce multiple robust solutions, we empirically show that it does not increase (but significantly decreases) the cost of search.

1 Introduction

Many problems in engineering, computer science, and management are naturally modeled as Constraint Satisfaction Problems (CSPs), which are, in general, in **NP**-complete. Solving these problems may require search. Symmetry has been exploited to improve the performance of search at least as far back as (Glaisher 1874); recently there has been a series of workshops on symmetry and CSPs (SymCon 2004). Our study focuses on the discovery and use of approximate symmetries during search that yield multiple, robust solutions. The symmetry relations we discuss are based on the notions of local value interchangeability (Freuder 1991), which group equivalent values of a variable in a bundle. Bundling refers to the mechanism of assigning such a bundle of values to a variable during search, yielding the generation of a *solution bundle*. A solution bundle is the set of solutions in the Cartesian product of the assigned domain bundles. *The primary goal of bundling is the generation of multiple robust solutions*. It was incorrectly assumed that static bundling (i.e., prior to search) and, a fortiori, dynamic bundling (i.e., during search), are too costly and not worthwhile. Beckwith et al. (2001) and Choueiry and Davis (2002) showed how to implement bundling at no more cost than search without bundling. They established theoretically this result for finding all solutions, and empirically for finding one solution. They also showed that dynamic bundling yields significantly larger solution bundles than static bundling while further reducing search cost. However, their work was limited to binary CSPs.

In this paper we introduce an efficient technique for bundling non-binary CSPs because the models of many real-life problems may have non-binary constraints[1]. We demonstrate the effectiveness of our technique on randomly generated problems in terms of the number of robust solutions found (primary goal) and the reduction of search cost (fortunate side effect). Our contributions are as follows:

1. An algorithm for computing the NI values of a CSP variable given a subset of the constraints defined on the variable regardless of their arities.
2. The integration of this mechanism with backtrack search, which we call dynamic bundling. And,
3. Experiments showing the benefits of dynamic bundling.

This paper is organized as follows. Section 2 states the motivations and background of our work. Section 3 describes the computation of NI values in the presence of non-binary constraints and its use in search with forward checking. Section 4 reports our experiments and analysis. Finally, Section 5 concludes this paper with directions for future research.

2 Motivation and Background

Beckwith et al. studied dynamic bundling in the context of binary CSPs (2001). They established that, when seeking all solutions, the number of constraint checks and the number of nodes visited by dynamic bundling may never exceed the corresponding numbers of search without bundling. Choueiry and Davis showed that those results hold empirically when seeking the first solution (2002). They concluded that dynamic bundling (primarily used for finding multiple, robust solutions) actually provides an effective means to improve search performance, drastically abating the peak cost of search at the phase transition. In Section 3.3 we explain this counter-intuitive result by the fact that *dynamic bundling is capable of bundling no-goods*, defined as partial

[1]The descriptions of workshops at IJCAI and ECAI on non-binary constraints stated that: "more and more attention is being paid to non binary constraints, mainly influenced by the growing number of real-life applications."

solutions that cannot yield complete solutions. Thus, dynamic bundling appears as a double-edged sword that not only produces robust solutions but also reduces thrashing during search.

Our goal here is to extend NI to non-binary CSPs and to establish the benefits of dynamic bundling in this context. In this paper we restrict ourselves to presenting the methods and evaluating them on randomly generated problems. More generally, we believe that the extension of local interchangeability to non-binary CSPs will be more useful than the original binary approach, which was advantageously used in case-based reasoning (Neagu and Faltings 2001) and local search (Petcu and Faltings 2003). Indeed, in (Lal and Choueiry 2004), we used the ideas presented here to design a novel query-join algorithm, reinforcing the impact of CP techniques on Databases and revealing new ways for supporting important functionalities in Databases such as query-size estimation.

2.1 Constraint satisfaction problems

A Constraint Satisfaction Problem (CSP) is defined by $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ where $\mathcal{V}$= $\{V_i\}$ is a set of variables, $\mathcal{D}$= $\{D_{V_i}\}$ the set of their respective domains, and $\mathcal{C}$ a set of constraints that restrict the acceptable combination of values for variables. Solving a CSP requires assigning a value to each variable such that all constraints are satisfied. The problem is in **NP**-complete in general. The *scope* of a constraint is the set of variables to which the constraint applies, and its *arity* is the size of this set. A constraint over the variables V_i, V_j, …, V_k is a set of tuples, subset of the Cartesian product of the domains of the variables in its scope: $C_{V_i,V_j,\ldots,V_k}$ = $\{(\langle V_i\ a_i\rangle, \langle V_j\ a_j\rangle, \ldots, \langle V_k\ a_k\rangle)\}$ where $a_i \in D_{V_i}$ and $\langle V_i\ a_i\rangle$ denotes a variable-value pair (vvp). We denote by NEIGHBORS(V_i) the set of variables that appear in the scope of the constraints that apply to V_i. We assume that the domains of the variables are finite.

A CSP is often represented as a graph, or constraint network, in which a node represents a variable and is labeled by the corresponding domain. A non-binary constraint is represented as a hyper-edge linking the nodes in the scope of the constraint. For sake of clarity, we represent a hyper-edge as another type of node connected to the variables in the scope of the constraint (see Figure 1 with the constraint given in Table 1).

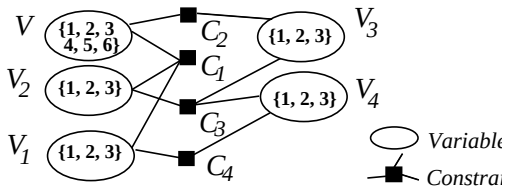

Figure 1: CSP.

We use the following parameters to assess the worst-case complexity of an algorithm applied to a CSP and for generating random instances:

- n number of variables,

Table 1: Constraint definitions.

C_1			C_2		C_3			C_4	
V	V_1	V_2	V	V_3	V_2	V_3	V_4	V_1	V_4
1	1	3	1	3	1	2	1	1	1
1	3	3	2	3	1	2	2	2	2
2	1	3	3	2	2	2	1	3	1
2	3	3	4	2	2	2	2		
3	1	1	4	2	3	1	1		
3	2	2	6	1					
4	1	1							
4	2	2							
5	3	2							
6	3	2							

- a maximum domain size,
- deg node degree,
- c_k number of constraints of arity k,
- $p_k = c_k / \binom{n}{k}$ constraint ratio of arity k, and
- t constraint tightness defined as the ratio of the number of disallowed tuples over the number of all possible tuples.

CSPs are typically solved using depth-first search with backtracking. In this paper, we use forward checking during search (FC) and order the variables dynamically according to the least domain heuristic. We denote the current variable by V_c, the set of future variables (i.e., uninstantiated variables) by $\mathcal{V}_f$, and the set of past variables (i.e., instantiated variables) by $\mathcal{V}_p$. At any point during search, the path from the root of the tree to the current variable is a set of variable-value pairs $\{\langle V_i\ a_i\rangle\}$ for the variables $V_i \in \mathcal{V}_p$ and their instantiations a_i. Forward checking on non-binary CSPs requires particular attention as we discuss in Section 3.2. As stated above, a *no-good* is any combination of variable-value pairs that cannot be extended to a consistent solution.

The performance of search is empirically evaluated by counting the number of constraint checks, the number of nodes visited, and the CPU time. Empirical studies of the performance of algorithms applied on CSPs are typically conducted in the area of the phase transition where the cost increases significantly around a critical value of an order parameter (Cheeseman *et al.* 1991).

2.2 Interchangeability & solution robustness

In its broadest sense, interchangeability allows one to recover one solution to a CSP from another (Freuder 1991). When solutions to a CSP are given, one can always define a mapping between the solutions such that one solution can be obtained from another without performing search. This is called *functional* interchangeability. Permutation of values across variables is called isomorphic interchangeability. We address here another restricted form of interchangeability: the interchangeability of values in the domain of a single variable. When values in the domain of a given variable are found interchangeable (i.e., equivalent), they can replace each other as an assignment to the variable in any solution

to the CSP, thus yielding robust solutions (Ginsberg *et al.* 1998; Choueiry and Davis 2002).

Definition 2.1 Neighborhood interchangeability (NI) (Freuder (1991)): *A value $a \in D_V$ is neighborhood interchangeable with a value $b \in D_V$ iff for every constraint C on V, a and b are consistent with exactly the same values.*

Algorithm 1 identifies the NI values for a variable V in $O(n \cdot a^2)$ by building a discrimination tree (DT) (Freuder 1991). Figure 3 shows DT(V_2) for the CSP in Figure 2.

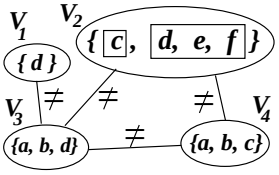

Figure 2: A binary CSP.

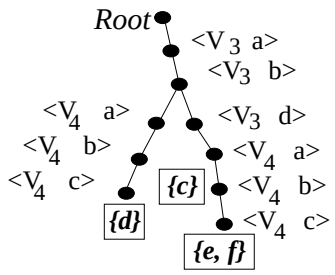

Figure 3: Partitioning D_{V_2}.

In this tree, the nodes represent variable-value pairs in the neighborhood of V_2. Some nodes are annotated with values from D_{V_2}, these annotations form a partition of D_{V_2}. All the variable-value pairs that appear in a path from the root of the tree to an annotation are consistent with the values appearing the annotation. It is important, in this procedure, that variables and values be ordered in a canonical way (e.g., lexicographical). For the CSP of Figure 2, values e and f are NI for V_2. If we had all the solutions of this CSP we would find that the values d, e, and f are interchangeable for V_2. Identifying such a situation may require finding all solutions to the CSP and hence is likely intractable.

2.3 Static and dynamic bundling

Benson and Freuder used NI to improve search (1992). A weaker form of NI, called *neighborhood interchangeability according to one constraint* (NI_C), was also used in search by Haselböck (1993). This search process yields solutions where some variables have a set of equivalent values, called a bundle. Both papers compute interchangeability sets *prior* to search, which corresponds to *static bundling*. Figure 4 shows a search tree for the example of Figure 2 without bundling (left) and with static bundling (center).

Freuder (1991) noticed that computing interchangeability *during* problem solving results in a weak type of interchangeability, *dynamic interchangeability*. Beckwith et al. (2001) and Choueiry and Davis (2002) showed how to recompute interchangeability partitions *during* search such that the resulting process, *dynamic bundling* (DynBndl), is always beneficial: it yields larger bundles and reduces the search effort. Figure 4 (right) shows the tree generated by dynamic bundling. The computational savings can be traced to:

1. bundling,
2. factoring out no-goods, and
3. reusing information from the discrimination tree for forward checking.

Further, they showed that, in comparison to dynamic bundling, static bundling is prohibitively expensive, particularly ineffective, and should be avoided.

The Cross Product Representation (CPR) (Hubbe and Freuder 1989) yields the same resulting bundles as dynamic bundling, but it requires more space and does not bundle no-goods. CPR necessarily visits more nodes than DynBndl, even though the difference is polynomially bounded.

Input: V
1 *current-node* ← *Root*, root of the discrimination tree
2 **for** *each value* $a \in D_V$ **do**
3 **for** *each variable* $V_j \in \text{NEIGHBORS}(V)$ **do**
4 **for** *each value* $x \in D_{V_j}$ *consistent with* a *for* V **do**
5 **if** *current-node* has a child node n_t with '$\langle V_j\ x \rangle$'
 then *current-node* ← n_t **else**
 Generate n_t a node with '$\langle V_j\ x \rangle$' and make it a child of *current-node*
6 *current-node* ← n_t
 end
 end
7 Add '$V, \{a\}$' to annotation of *current-node*
8 *current-node* ← *Root*
 end
end
Output: *Root*

Algorithm 1: Algorithm for building DT(V).

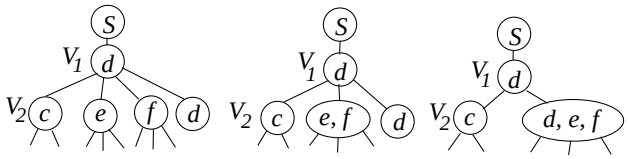

Figure 4: Search with no, static, and dynamic bundling.

3 Dynamic Bundling for non-binary CSPs

In this section we first a mechanism for computing NI values in the presence of non-binary constraints. Then, we discuss how non-binary constraints are updated for FC. Finally, we describe the integration of the computation of interchangeability with search, which we call dynamic bundling. We also illustrate the compaction of the solution space and the factoring of no-goods.

3.1 NI for non-binary constraints

A direct application of Algorithm 1 to non-binary CSPs may yield incorrect results. Consider the CSP defined in Figures 5 and 6. It is clear from the definition of C_1 that $\langle V\ x\rangle$

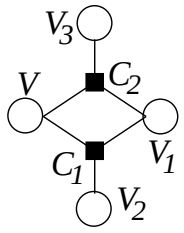

Figure 5: CSP.

C_1			C_2		
V	V_1	V_2	V	V_1	V_3
x	a	1	x	a	1
x	b	2	x	b	2
x	c	3	x	c	3
x	d	1	y	a	1
y	a	1	y	b	2
y	b	2	y	c	3
y	c	3			

Figure 6: C_1, C_2.

and $\langle V\ y\rangle$ are consistent with unequal sets of tuples and thus are not interchangeable. We show that a direct application of Algorithm 1 detects them as interchangeable because they are consistent with the same variable-value pairs in the neighborhood of V. Indeed, Line 4 of Algorithm 1 requires checking consistency according to *all* the constraints defined on V. The values of V_1 consistent with $\langle V\ x\rangle$ given C_1 and C_2 are {a, b, c} (i.e., {a, b, c, d} given C_1 and {a, b, c} given C_2). Similarly the values for V_2 and V_3 consistent with $\langle V\ x\rangle$ given the constraints are {1, 2, 3} and {1, 2, 3}, respectively. Finally, $\langle V\ y\rangle$ is found to be consistent with the same values, yielding the DT shown in Figure 7, where x, y are found interchangeable for V. The overlapping scopes of constraints make the direct application of Algorithm 1 to the non-binary case unfit.

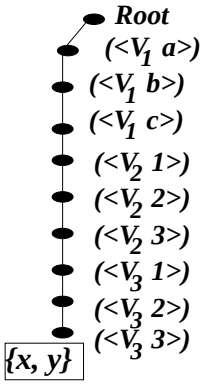

Figure 7: DT(V).

Our technique is based on building a separate discrimination tree for *each* of the deg constraints defined on V. We call such a tree a *non-binary discrimination tree* (nb-DT). The NI sets of V and the domains of the neighboring variables consistent with each NI set are then derived from 'overlapping' the individual nb-DTs. Below, we introduce two processes. The first partitions the domain of V by building then combining the relevant nb-DTs; and the second determines the values of the neighboring variables consistent with each set in the partition. These two processes are used in dynamic bundling (Section 3.3) for computing the bundles of the current variable (Process 1) and for forward checking (Process 2).

Process 1: Computing a domain partition. First, an nb-DT is created for each one of the deg constraints on V using Algorithm 2. This algorithm is similar to Algorithm 1 except that it operates only on one constraint C and compares each value of V with a tuple of C.

```
   Input: V, C
1  current-node ← Root, root of the discrimination tree
2  S ← SCOPE(C) \ {V}
3  for every value v ∈ D_V do
4      for every tuple t ∈ C | σ_{V=v}(t) exists do
5          if current-node has a child node n_t equal to π_S(t)
           then current-node ← n_t else
               Generate n_t a node with π_S(t) and make it a
               child of current-node
6              current-node ← n_t
           end
       end
7      Add '⟨V, {v}⟩' to the annotation of current-node
8      current-node ← Root
   end
   Output: Root
```

Algorithm 2: Algorithm for building nb-DT(V, C).

Line 4 of Algorithm 2 replaces Line 3 and 4 of Algorithm 1. σ and π correspond respectively to the selection and projection operators in relational algebra. The worst-case time complexity of Algorithm 2 is linear in the size of the constraint, which depends on the variable domain sizes and the tightness and arity of the constraint. The cost of building deg nb-DTs is $O(deg \cdot a^{k+1} \cdot (1-t))$. Figures 8 and 9 show, for the example of Figure 1 and Table 1, the nb-DTs for V given C_1 and C_2, respectively.

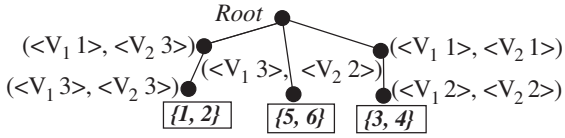

Figure 8: nb-DT(V, C_1).

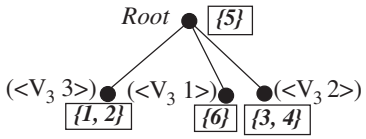

Figure 9: nb-DT(V, C_2).

Second, for each tree, we collect the annotations and the path where they appear. We traverse the tree from the root to each annotation A_i and construct P_i by collecting the nodes

on the path. We form a list l_i = (P_i, A_i) of the particular path and the corresponding annotation, and a list $L_j = \{l_i\}$ of these lists for each nb-DT. In the example of Figures 8 and 9:

1. For the nb-DT of C_1, $L_1 = (l_1, l_2, l_3)$ with:
   - l_1 = ((($\langle V_1\ 1\rangle$, $\langle V_2\ 3\rangle$), ($\langle V_1\ 3\rangle$, $\langle V_2\ 3\rangle$)), {1, 2}),
   - l_2 = ((($\langle V_1\ 3\rangle$, $\langle V_2\ 2\rangle$)), {5, 6}),
   - l_3 = ((($\langle V_1\ 1\rangle$, $\langle V_2\ 1\rangle$), ($\langle V_1\ 2\rangle$, $\langle V_2\ 2\rangle$)), {3, 4}).
2. For the nb-DT of C_2, $L_2 = (l_4, l_5, l_6, l_7)$ with
   - l_4 = ((($\langle V_3\ 3\rangle$)), {1, 2}),
   - l_5 = ((($\langle V_3$ `nil`$\rangle$)), {5}),
   - l_6 = ((($\langle V_3\ 2\rangle$)), {3, 4}),
   - l_7 = ((($\langle V_3\ 2\rangle$)), {6}).

We collect these lists in $L = (L_1, L_2, \ldots, L_d)$.

Third, we compute the partition of D_V by intersecting the annotation A_i from each tree using Algorithm 3 with L and V as input parameters. The worst-case time complexity of this algorithm is $O(deg^2 \cdot a^4)$.

Input: L, V
1 *dom-values* ← domain of V
2 *partitioned-domain* ← `nil`
3 **for** *every value* v *remaining in* dom-values **do**
4 *select-path+annot* ← An l_i from every $L_j \in L$ for which $v \in$ ANNOTATION(l_i)
5 *annotation* ← Intersect annotations in the *select-path+annot*
6 Add *annotation* to *partitioned-domain*
7 *dom-values* ← *dom-values* \ *annotation*
end
Output: *partitioned-domain*

Algorithm 3: Intersecting annotations.

For the example of Figures 8 and 9, the domain of V is partitioned as {{1, 2}, {3, 4}, {5}, {6}}. We denote by E_i an element of this partition, where E_i is a set of equivalent values of V given the constraints that apply to it.

To reduce the cost of Process 1, we have implemented a mechanism that automatically 'switches off' some operations when it becomes clear that all sets in the partition of D_V are necessarily singletons. Such an opportunity occurs (1) when an nb-DT of V results in annotations exclusively made of singleton elements (see Algorithm 2); and (2) when the intersection of the annotations returns singletons (see Algorithm 3).

Process 2: Neighboring values consistent with an E_i. This process computes the values in the neighborhood of V that are consistent with each equivalence class E_i using the nb-DTs built in Process 1. For a given E_i, we identify the paths $\{P_i\}$ in each nb-DT such that $E_i \subseteq A_i$. Then, for each $X \in$ NEIGHBORS(V), we project each path P_i on X. Intersecting the results of the projections yields the subset of D_X that is consistent with E_i. (We use this information below to update D_X by forward checking after assigning E_i to V.)

3.2 Forward checking with non-binary constraints

Independently of bundling, two issues arise when applying FC to non-binary CSPs: (1) choosing the subset of constraints to account for, and (2) updating their definitions to reflect past instantiations and domain prunings. We adopt the strategy called `nFC2` (Bessière *et al.* 2002), where the constraints considered are the ones that apply to the current variable V_c and at least one future variable. The update of a non-binary constraint C according to past instantiations amounts to intersecting the definition of C with the Cartesian product of the (updated) domains of V_c and future variables. We propose here a more efficient implementation that uses a linear number of selection and projection operations. Let SCOPE(C) = $\mathcal{V}_a \cup \{V_c\} \cup \mathcal{V}_b$, where $\mathcal{V}_a \subseteq \mathcal{V}_p$ and $\mathcal{V}_b \subseteq \mathcal{V}_f$. The domains of variables in $\{V_c\} \cup \mathcal{V}_b$ might have already been filtered by FC, and certain tuples in C might have become invalid. Thus, we need to select the tuples of C that have survived the filtering by FC after the instantiation of past variables. The selected tuples must satisfy: (1) $\langle V_i\ a_i\rangle$ for $V_i \in \mathcal{V}_a$ and a_i the instantiation of V_i; and (2) $a_j \in D_{V_j}$ for $V_j \in \{V_c\} \cup \mathcal{V}_a$, where D_{V_j} are filtered domains. We denote this operation $\sigma^{FC}_{\mathcal{V}_p}(C)$. In order to compute the updated constraint, we project $\sigma^{FC}_{\mathcal{V}_p}(C)$ on $\{V_c\} \cup \mathcal{V}_b$,

$$C' = \pi_{\{V_c\} \cup \mathcal{V}_b}(\sigma^{FC}_{\mathcal{V}_p}(C)). \quad (1)$$

The way non-binary FC (without bundling) is implemented significantly affects the number of constraint checks and CPU time spent to solve a CSP. If a given instantiation for V_c is later discarded when search backtracks to V_c, then the same updated constraint C' of Equation (1) can be reused because it is valid for all values in D_{V_c}. Hence, we choose to store each C' associated with V_c. Note that by doing so we level the playing field for the two algorithms being compared (i.e., DynBndl and FC). Thus, our empirical results reflect the gain due purely to bundling and exclude the benefits gained from the additional nb-DT data structure.

3.3 Dynamic bundling

DynBndl operates by assigning a bundle to V_c and propagating the effect of this decision on the future variables. The bundles of V_c are obtained by applying Process 1 of Section 3.1 using the constraints on V_c determined by `nFC2`. Each constraint passed to Algorithm 2 is computed using Equation (1). The effects of this instantiation are then propagated using Process 2 of Section 3.1. Figure 10 (left) shows partially the search tree explored by FC for the example in Figure 1 and Table 1 with variable ordering {V, V_1, V_2, V_3, V_4}. Figure 10 (right) shows the one explored by DynBndl. This example illustrates two situations that result in performance gain: bundling of no-goods and bundling of solutions. When DynBndl assigns {1,2} to V, {1,3} to V_1 and {3} to V_2, D_{V_3} is annihilated after visiting 3 nodes, whereas FC visits 10. This situation illustrates the gains of no-good bundling. When DynBndl next assigns {3,4} to V, search succeeds and DynBndl yields 2 solutions, while FC yields a

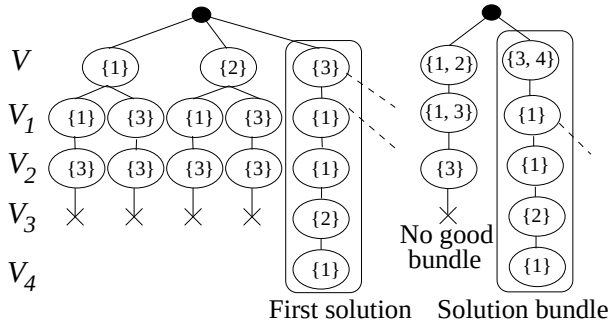

Figure 10: Search tree with FC (left) and with DynBndl (right).

single solution. More generally, under the same variable and value ordering, DynBndl visits no more nodes than FC.

The use of a MAC-like, full lookahead schema (Sabin and Freuder 1994) necessarily performs a better filtering of the domains of $\mathcal{V}_f$. While full lookahead may increase the number of constraint checks, it may yield 'fatter' solution bundles and reduce of number of nodes visited. Note that DynBndl, while it partitions the set of solutions (i.e., every solution appears in exactly one bundle), does not guarantee that the size of the solution bundle is maximal (i.e., the size of the bundle cannot be increased (Lesaint 1994)).

Since building an nb-DT requires the enumeration of a constraint's tuples, our technique may be cumbersome to use in presence of one or more global constraints that typically have an exponential size. We can still apply our techniques to the problem from which we have removed the bulky global constraints, and then use the discarded constraints to discriminate among the solutions produced in a solution bundle. This approach is reminiscent of multi-dimensional CSPs by (Freuder and Sabin 1997).

4 Evaluation

Below we discuss the choice of test problems and our experimental set-up, then summarize the results of our experiments on evaluating the effectiveness of bundling in terms of returning multiple solutions and reducing the cost of search.

The first experiment aims at demonstrating the effect of varying constraint tightness (see Section 4.3). It shows that dynamic bundling is able to find multiple solutions even in the area of the phase transition, where it is also most effective in reducing the cost of search. The second experiment focuses on the area of the phase transition and investigates the effect of varying domain size (see Section 4.4). It shows that increasing domain size increases the benefit of dynamic bundling.

4.1 Choice of test problems

Neighborhood interchangeability aims at detecting equivalent values in the domain of a given CSP variable. It does not pretend to uncover permutations of values across variables, which is isomorphic interchangeability and is the focus of most work on symmetry in CSPs. One can expect neighborhood interchangeability, and its weaker version used in dynamic bundling, to be useful in real-world applications where domain redundancy exists or appears during search. This is not the case of the benchmark problems used for symmetric CSPs. While looking for (strong or weak) NI sets is cost effective and should be always attempted, no technique can find multiple robust solutions in permutation problems where there are exactly as many variables as there are values.

The primary practical advantage of bundling is the production of robust solutions, where any value in a bundle for a given variable can replace any other value in the bundle should the former become unavailable or undesirable. The practical usefulness of neighborhood interchangeability was established in case-based reasoning (Neagu and Faltings 2001), nurse scheduling (Weil and Heus 1998), and databases (Lal and Choueiry 2004). For example, in (Lal and Choueiry 2004) we reduced the size of a query result on a real-world database by 54% (yet storing the same information). While we still need to validate our approach on real-world applications, in this paper, we focus on introducing the techniques and their implementation and test our algorithms on randomly generated CSPs. Even though such problems lack the redundancy one expects to find in real-world applications (which makes them particularly amenable to bundling), our experiments show that dynamic bundling effectively yields multiple robust solutions.

4.2 Experiment design and set-up

In our experiments, we compared the performance of backtrack search with forward checking (FC) and backtrack search with forward checking and dynamic bundling (DynBndl) using the least domain (respectively, the least number of bundles in a domain) heuristic for dynamic variable ordering. We describe a non-binary CSP with the tuple $\langle n, a, p_2, c_3, c_4, t\rangle$, where:

- n the number of variables,
- a the domain size,
- p_2 the ratio of binary constraints, c_3 and c_4 the number of ternary and quaternary constraint respectively,
- and t the constraint tightness.

We used a uniform random generator. While varying t, we tested the following 16 combinations:

- n=$\{20, 30\}$,
- a=$\{10, 15\}$, and
- Constraint ratio (CR)=$\{$CR1, CR2, CR3, CR4$\}$ defined in Table 2.

We included datasets that we expect to be less favorable to bundling (e.g., n = 20, a = 10, CR = CR4) and more favorable to bundling (e.g., n =30, a = 15, CR = CR1) given the larger domain size and the smaller constraint ratio.

We measured the size of the first solution bundle (FBS), CPU time, number of nodes visited NV, and number of constraint checks (CC). FBS is an indicator of the effectiveness of bundling in returning multiple solutions. CPU time, NV, and CC are indicators of the cost of problem solving.

Table 2: Constraint ratios of random instances.

Constraint Ratio (CR)	p_2	c_3	c_4
CR1	0.25	3	2
CR2	0.25	6	5
CR3	0.4	3	2
CR4	0.4	6	5

Following the guidance of expert statisticians, we applied a log transformation on the numerical results to reduce high variance and fit the chosen statistical model. We used ANOVA (ANalysis of VAriances) to study the interaction of DynBndl and FC while varying t (Rees 2001). At every t, we estimated the difference in the mean values of a given measurement and the confidence intervals of these values using the t-distribution. The ANOVA results indicate whether there is a statistically significant difference in the mean values of DynBndl and FC. We chose a size sample of 1000 instances after running a pilot experiment with 10000 instances and identifying where the plot of moving averages stabilized. Below we report the results of varying tightness then the effect of increasing a.

4.3 Varying tightness

Figure 11 shows the CPU time (lower curves) and number of nodes visited (upper curves) for FC and DynBndl. It also lists in a table the first bundle size for various values of constraint tightness. Quite expectedly, the largest FBS occurs at low tightness values, however, DynBndl finds non-singleton solution bundles also well into the area of the phase transition. Below, we distinguish three regions: low tightness, around the cross-over point, and high tightness.

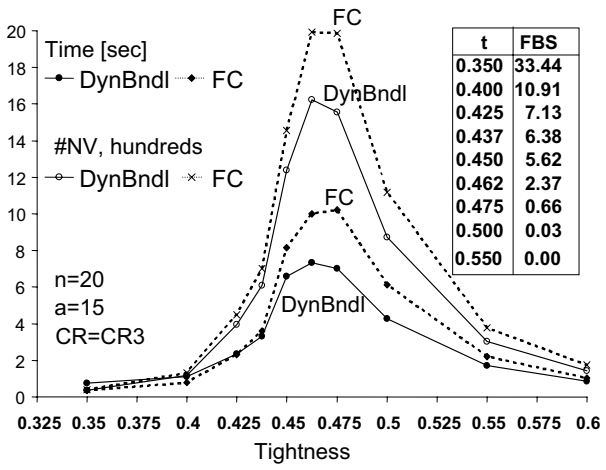

Figure 11: CPU time and #NV.

At low tightness ($t \leq 0.425$), the first solution (bundle) is found without much backtracking. The benefit of DynBndl here is the large FBS. For example, we have FBS=33 at t=0.350. (Not shown here, for n=30, d=15, CR1, t=0.350, we have FBS=2254.7 (Lal 2005).) The benefit of bundling no-goods is not yet visible.

While the cost of computing the bundles is visible (the constraint definitions are large), the overhead can be ignored given the short total time for solving the problems.

At t=0.425 ANOVA shows no significant difference between the CPU time of DynBndl and FC: the overhead of computing the bundles is compensated by the bundling of no-goods.

Around the cross-over point ($0.425 < t \leq 0.500$), DynBndl still yields multiple solutions (e.g., FBS=5 at t=0.450 and FBS=2.3 at t=0.462).

Further, bundling of no-goods by DynBndl becomes prevalent yielding the maximum amount of savings in NV, CC, and CPU time.

ANOVA indicates significant improvement of DynBndl over FC across the entire region.

For high tightness ($0.500 < t$), forward checking effectively detects that most of the CSPs are not solvable early on in the search process, thus reducing NV.

The overhead of bundling shows up again however ANOVA indicates that DynBndl and FC are still comparable at t=0.600.

4.4 Effect of increasing domain size

Table 3 shows, in the phase-transition region, the average improvement of FBS and CPU time when increasing the domain size. We compute the improvement of a measurement X as:

$$\text{Improvement}(X) = \frac{X(\text{FC}) - X(\text{DynBndl})}{X(\text{FC})}.$$

In short, increasing domain size increases the value of FBS and also the CPU time savings by DynBndl. While the cost of computing NI sets increases with the domain size (i.e., $O(deg \cdot a^{k+1} \cdot (1-t))$, see Section 3.1), our experiments show that the savings due to the no-good bundling offsets this cost increase. This feature is especially promising in the context of application to databases where large domain sizes are typical.

Table 3: Increasing a (n=30) around phase transition.

CR	FBS		CPU improvement	
	a=10	a=15	a=10	a=15
CR1	5.55	11.93	33.35%	34.32%
CR2	5.01	5.52	28.58%	33.01%
CR3	3.55	4.95	29.82%	31.66%
CR4	1.23	1.43	28.45%	31.65%

5 Conclusions and future work

This paper describes how to compute NI sets and discusses their use for dynamic bundling in non-binary CSPs. In the future, we propose to evaluate the effectiveness of our techniques on custom document assembly (Purvis 2002) and

query optimization using materialized views in databases. We believe that the area of databases, where the constraints are typically defined in extension as (very large) tables, is particularly well-suited for the use of dynamic bundling techniques as we demonstrated in (Lal and Choueiry 2004) and discussed in more detail in (Lal 2005).

Acknowledgments. The authors are grateful to Nic Wilson and Steve Prestwich for their comments, and to Bradford Danner for his help with the statistical analysis. This work was supported by the Maude Hammond Fling Faculty Research Fellowship, CAREER Award #0133568 from the National Science Foundation, and Science Foundation Ireland under Grant 00/PI.1/C075. The experiments were conducted on the Research Computing Facility of UNL.

References

Beckwith, A.M.; Choueiry, B.Y.; and Zou, H. 2001. How the Level of Interchangeability Embedded in a Finite Constraint Satisfaction Problem Affects the Performance of Search. In *Australian Joint Conf. on AI*, LNAI 2256. 50–61.

Benson, B.W. and Freuder, E.C. 1992. Interchangeability Preprocessing Can Improve Forward Checking Search. In *Proc. of ECAI*. 28–30.

Bessière, C.; Meseguer, P.; Freuder, E.C.; and Larrosa, J. 2002. On Forward Checking for Non-binary Constraint Satisfaction. *Artificial Intelligence* 141 (1-2):205–224.

Cheeseman, P.; Kanefsky, B.; and Taylor, W.M. 1991. Where the Really Hard Problems Are. In *Proc. of IJCAI*. 331–337.

Choueiry, B.Y. and Davis, A.M. 2002. Dynamic Bundling: Less Effort for More Solutions. In *Proc. of SARA*, volume 2371 of *LNAI*. Springer. 64–82.

Freuder, E.C. and Sabin, D. 1997. Interchangeability Supports Abstraction and Reformulation for Multi-Dimensional Constraint Satisfaction. In *Proc. of AAAI*. 191–196.

Freuder, E.C. 1991. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *Proc. of AAAI*. 227–233.

Ginsberg, M.L.; Parkes, A.J.; and Roy, A. 1998. Supermodels and Robustness. In *Proc. of AAAI*. 334–339.

Glaisher, J.W.L. 1874. On the Problem of the Eight Queens. *Philosophical Magazine, series 4* 48:457–467.

Haselböck, A. 1993. Exploiting Interchangeabilities in Constraint Satisfaction Problems. In *Proc. of IJCAI*. 282–287.

Hubbe, P.D. and Freuder, E.C. 1989. An Efficient Cross Product Representation of the Constraint Satisfaction Problem Search Space. In *Proc. of AAAI*. 421–427.

Lal, A. and Choueiry, B.Y. 2004. Constraint Processing Techniques for Improving Join Computation: A Proof of Concept. In *Proc. of the Symposium on Constraint Databases*, LNCS 3074. 149–167.

Lal, Anagh 2005. Neighborhood Interchangeability for Non-Binary CSPs & Application to Databases. Master's thesis, Constraint Systems Laboratory, Department of Computer Science & Engineering, University of Nebraska-Lincoln, Lincoln, NE.

Lesaint, D. 1994. Maximal Sets of Solutions for Constraint Satisfaction Problems. In *Proc. of ECAI*. 110–114.

Neagu, N. and Faltings, B. 2001. Exploiting Interchangeabilities for Case Adaptation. In *International Conference on Case-Based Reasoning (ICCBR 01)*, volume 2080 of *LNCS*. Springer. 422–436.

Petcu, A. and Faltings, B. 2003. Applying Interchangeability Techniques to the Distributed Breakout Algorithm. In *Proc. of IJCAI*. 1381–1382.

Purvis, L. 2002. A Genetic Algorithm Approach to Automated Custom Document Assembly. In *Proc. of the Intelligent Systems Design and Applications Conference*. 131–136.

Rees, D.G. 2001. *Essential Statistics*. Chapman and Hall.

Sabin, D. and Freuder, E.C. 1994. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proc. of ECAI*. 125–129.

SymCon, 2004. The 4th International Workshop on Symmetry and Constraint Satisfaction Problems. http://www.dis.uu.se/SymCon04/.

Weil, G. and Heus, K. 1998. Eliminating Interchangeable Values in the Nurse Scheduling Problem Formulated as a Constraint Satisfaction Problem. In *Workshop on Constraint-Based Reasoning (FLAIRS 95)*.