Dynamic Detection and Exploitation of Value Symmetries for Non-Binary Finite CSPs

Anagh Lal and Berthe Y. Choueiry

Constraint Systems Laboratory
Department of Computer Science and Engineering
University of Nebraska-Lincoln
{alal|choueiry}@cse.unl.edu

Abstract. Bundling the symmetrical values in the domain of variables of a Constraint Satisfaction Problem (CSP) as the search proceeds is an abstraction mechanism that yields a compact representation of the solution space. In [3, 9], we showed empirically that dynamic bundling during backtrack search for finding the first or all solutions of binary random CSPs is always beneficial. In this paper, we describe a method[1] to perform dynamic bundling during search in non-binary CSPs using a data structure called a non-binary discrimination Tree (NB-DT). We conduct experiments using backtrack search with forward checking and dynamic variable ordering, and compare the performance of solving random non-binary CSPs with and without bundling. We show that the benefits of dynamic bundling encountered in binary CSPs continue in non-binary problems. We observe the phase transition phenomenon for non-binary CSPs and study how dynamic bundling performs in the phase transition region. We also describe a generator of non-binary random CSP instances that guarantees the existence of a solution and can generate constraints of any arity.

1 Introduction

Many problems in engineering, computer science, and management are naturally modeled as Constraint Satisfaction Problems (CSPs), which are, in general, **NP**-complete. Search remains the ultimate mechanism for solving these problems. Glaisher [14], Puget [20], Ellman [12] and many others proposed to exploit *declared* symmetries specific to a class of problems to improve the performance of search. The majority considered *exact* symmetries only, but Ellman also considered necessary and sufficient *approximations* of symmetry relations. While the above approaches focused on declared symmetry, this paper focuses on enhancing the performance of search by *dynamically discovering* and exploiting symmetries inherent to a particular instance of a problem. The symmetry mechanisms we study are based on the notions of local value interchangeability of Freuder [13]

[1] The algorithms described in this paper and their application to databases are the subject of a pending patent.

and domain bundling of Haselböck [15], which groups in a bundle (or equivalence class) the interchangeable values in the domain of a variable. Objections were raised that bundling mechanisms, applied statically (i.e., prior to search) or dynamically (i.e., during search), are too costly and not worthwhile when one is seeking one solution to a CSP. In [3, 9], we showed how to implement bundling to reduce drastically the search effort and yield *multiple and robust solutions for less effort than needed to find a single solution*. (This holds theoretically for finding all solutions, and empirically for finding one solution.) We also showed that dynamic bundling is significantly less expensive and more effective than static bundling. Our investigations were limited to binary CSPs.

Although most of the research in constraint satisfaction is performed on binary CSPs, many real-life problems are more 'naturally' modeled as non-binary CSPs. Because it is always possible in principle to reduce a non-binary finite CSP to a binary one [21, 1], the focus on binary constraints has so far been tolerated by the research community. Research on non-binary constraints is still in its infancy and the traditional attitudes on this issue are now being challenged [7]. In this paper we show how dynamic bundling can be done for non-binary constraints and demonstrate its advantage on toy and randomly-generated problems. While one expects real-world problems to exhibit redundancy, which is particularly amenable to bundling, it is reasonable to expect toy and randomly generated problems to lack the type of symmetry relations we are looking for, and thus resist bundling. We show that bundling remains beneficial even under these unfavorable conditions. Our contributions are as follows:

1. We introduce an algorithm for partitioning the domain of a CSP variable into equivalence classes given any subset of the constraints that apply to the variable regardless of their arities.
2. We integrate this mechanism with backtrack search for solving the CSP.
3. We introduce a generator of non-binary CSP instances that guarantees the existence of a solution and can handle constraints of any arity. To the best of our knowledge, this is the first such generator. It is based upon, but improves, the generators of [5, 23]. Because pre-processing techniques quickly detect the inconsistency of most random instances with tight constraints, guaranteeing the existence of a solution in these instances makes (1) bundling likely less effective and (2) the evaluation of search algorithms more meaningful.
4. We carry out extensive experiments that demonstrate the benefits of dynamic bundling (in terms of number of nodes visited, number of constraint checks, and CPU time) for finding one solution and all solutions.

This paper is organized as follows. Section 2 states the motivations and background of our work. Section 3 shows how to bundle the domain of a variable in presence of non-binary constraints. Section 4 discusses how to integrate bundling in backtrack search using non-binary forward-checking (FC) `nFC2` [6] to solve non-binary CSPs. Section 5 introduces a generator of random non-binary CSP instances that guarantees the existence of a solution. Section 6 reports our experiments and analysis. Finally, Section 7 concludes this paper and gives future directions for research.

2 Motivation and Background

In [2, 9] we established that dynamic bundling is guaranteed never to be costlier than no-bundling when seeking all solutions, and demonstrated empirically that this result also holds when seeking the first solution. These results are important in our opinion because they prove that dynamic bundling (which is used to get multiple, robust solutions) is actually beneficial for reducing the search effort and drastically reduces the peak cost of search at the phase transition. This counter-intuitive result can be explained by the fact that *dynamic bundling is capable of bundling no-goods as well as partial solutions*, and is thus a double-edged sword that reduces thrashing during search. Our goal here is to show that the benefits of dynamic bundling continue in non-binary CSPs and that dynamic bundling still prevails in the region of the phase transition. To this end, we describe a technique for partitioning the domain of a CSP variable in the presence of non-binary constraints and integrate this technique with backtrack search.

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 $C_{V_i,V_j,\ldots,V_k} = \{(\langle V_i\ a_i\rangle, \langle V_j\ a_j\rangle, \ldots, \langle V_k\ a_k\rangle)\}$ such that $a_i \in D_{V_i}$, $a_j \in D_{V_j}$, …, $a_k \in D_{V_k}$. *In this paper, we assume that the domains of the variables are finite and that the acceptable tuples can be enumerated in polynomial time in the size of the problem.* Solving a CSP requires assigning a value to each variable such that all constraints are simultaneously satisfied and the problem is, in general, **NP**-complete. A CSP is often represented by a graph. In this graph, nodes represent variables and are labeled by the respective domains. Edges linking two nodes represent constraints that apply to these corresponding variables. The *scope* of a constraint is the set of variables to which the constraint applies, and its *arity* is the size of this set. Non-binary constraints are represented as hyper-edges in the constraint network. For the sake of clarity, we choose to represent such a hyper-edge as another type of node that is linked to the variables in the scope of the constraint as shown in Fig. 1. In the CSP of Fig. 1 each variable has a domain of {1, 2, 3} and the constraints are shown in Fig 2. We use this CSP in Section 3 to show how to partition the domain of V given the constraints that apply to it. CSPs are typically solved using depth-first search. In this paper, we study backtrack search (BT) with forward checking (FC) and dynamic variable ordering. We use the heuristic that orders variables according to the minimum ratio of the domain size over the degree (which is the number of adjacent variables in the constraint network)[2]. Since we apply this heuristic dynamically, we denote it `DDD`. Depth-first search proceeds by choosing a variable (the current variable V_c), and instantiating it, that is assigning it a value that is taken from its domain.

[2] For non-binary CSPs, we compute the degree only once, prior to search, in order to avoid expensive computations during search.

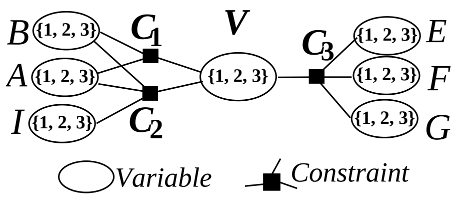

Fig. 1. *Example of a non-binary CSP.*

C_1		
A	B	V
1	2	1
1	2	2
2	2	2
1	2	3
2	3	1
2	3	3

C_2			
A	B	I	V
3	1	2	3
1	3	3	2
1	3	3	1
2	1	1	1
2	1	1	2

C_3			
E	F	G	V
2	3	1	3
2	3	1	1
1	2	3	1
1	2	3	3
3	2	1	2
2	1	1	2

Fig. 2. *Constraint tables.*

The variable and the value a assigned to it define a variable-value pair (vvp), which we denote as $\langle V_c \ a \rangle$. FC propagates the effect of this instantiation over the set of all uninstantiated variables, which we call future variables and denote as $\mathcal{V}_f$. The current variable V_c is then added to the set of instantiated variables, which we call past variables and denote as $\mathcal{V}_p$. If the instantiation does not wipe out the domain of any variable in $\mathcal{V}_f$, search proceeds to the next variable based on the variable-ordering schema chosen. Otherwise, the instantiation is revoked, its effects are undone, and an alternative instantiation to the current variable is attempted. When all alternatives fail, search backtracks to the previous level in the tree. The process repeats until one or all solutions are found. At any point during search, the path from the root of the tree to the current variable is a set of vvps $\{\langle V_i \ a_i \rangle\}$ for the variables V_i in $\mathcal{V}_p$ and their instantiation a_i. We denote by $\mathcal{P}_{\mathcal{V}_p}$ the induced CSP formed by the set $\{\langle V_f \ D_{V_f} \rangle\}$ of future variables and their respective filtered domains. Search on non-binary CSPs proceeds as described above. However, FC for non-binary CSPs requires particular attention. Also, counting the number of constraint checks during search (as a measure of the cost of search) is slightly different. A constraint check is more expensive in a non-binary CSP than in a binary one. Section 4 addresses these issues.

2.2 Interchangeability

Interchangeability is about finding redundant solutions in a CSP. When a CSP has more than one solution, one can define a mapping between the solutions such that if the mapping is known, one solution can be obtained from another without performing search. In the broadest sense, this is *functional* interchangeability proposed by Freuder [13]. We address here a restricted type of interchangeability: the interchangeability of the values in the domain of a single variable. This cannot detect isomorphic interchangeability such as permutation. Below we recall simplified forms of interchangeability and show in Fig. 3 how they relate.

Definition 1. Full interchangeability (FI)*: A value a in the domain of variable V is interchangeable with value b in the same domain iff every solution to the CSP that involves a remains a solution when b is substituted for a, and vice versa.*

In other words, two values of the variable V are fully interchangeable when the only difference between two solutions of a CSP is the value of V itself. Checking all the solutions of the CSP in Fig 4 we find that the values d, e, and f are fully interchangeable for V_2. The computation of full interchangeability may

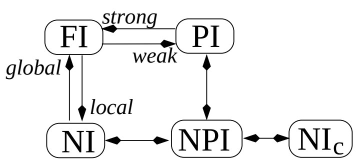

Fig. 3. *Some types of interchangeability and their relationship.*

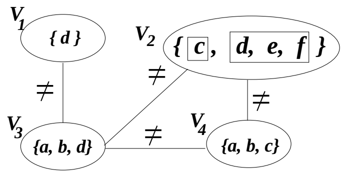

Fig. 4. *Partitioning the domain of V_2 into equivalence classes.*

require finding all solutions and hence is likely to be intractable and impractical in practice. Freuder [13] also identified a form of *local* interchangeability, called *neighborhood interchangeability*, which can be computed efficiently and is a sufficient approximation of full interchangeability. Freuder [13] provided an efficient algorithm, the discrimination tree (DT), for computing the partitions of a domain into equivalence classes based on neighborhood interchangeability.

Definition 2. Neighborhood interchangeability (NI)*: A value a in the domain of variable V is neighborhood interchangeable with a value b in the same domain iff for every constraint C incident to V, a and b are consistent with exactly the same values:* {x | (a, x) satisfies C} = {x | (b, x) satisfies C}.

Neighborhood interchangeability is a sufficient, but not necessary condition for full interchangeability. Indeed, in Fig. 4, only values e and f are NI for V_2 whereas values e, f, and d are FI for V_2. Both full interchangeability and neighborhood interchangeability do not permit variables other than in the selected variable V in the CSP to change. *Partial interchangeability* is a *weaker* kind of interchangeability, based on the idea that when a value for V changes, values for other variables may also differ among themselves but be fully interchangeable with respect to the rest of the CSP. We introduce a *boundary of change*, $\mathcal{S}$, within which we permit change. Partial interchangeability, like full interchangeability, is a global form of interchangeability. We can localize partial interchangeability in the same way we localized full interchangeability, by only considering those variables whose constraints cross the boundary of $\mathcal{S}$. (These are the variables in the neighborhood of $\mathcal{S}$.) This is called *neighborhood partial interchangeability* (NPI) and was introduced in [10]. Note that NPI is a sufficient but not necessary condition for NI. In [10] we showed how to extend the discrimination tree of [13] into the *joint discrimination tree* (JDT) to partition the domain of a variable V into sets of values that are NPI for V. NPI has since been applied in case-based reasoning [18] and local search [19].

Freuder [13] noticed that computing interchangeability *during* problem-solving results in yet another weak type of interchangeability, *dynamic interchangeability*. In [2, 9] we introduced dynamic NPI (DNPI), a weaker version of NPI based on the dynamic computation of NPI sets during search. DNPI holds only along the particular path following by search and thus yields larger partitions.

Definition 3. Dynamic NPI*: Given a variable ordering in a backtrack search that integrates any kind of lookahead scheme, DNPI is obtained by partitioning domain of the current variable, V_c, obtained by the JDT of V_c with $\mathcal{S} = \mathcal{V}_p \cup \{V_c\}$, where $\mathcal{V}_p$ is the set of past variables in the search tree.*

Neighborhood interchangeability has been tested in the context of search by Benson and Freuder [4]. A weaker form of neighborhood interchangeability, which we call *neighborhood interchangeability according to one constraint* (NI_C), by Haselböck, was also used in search [15]. Both [4, 15] compute interchangeability sets *prior* to search. We call such strategies *static bundling*. Fig. 5 shows a search tree for the example of Fig. 4 without bundling (left) and with static bundling (right). In [2, 9], we established that recomputing interchangeability

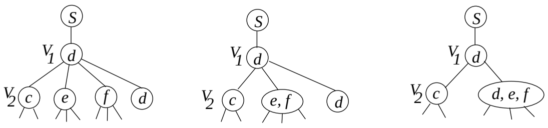

Fig. 5. *Search tree.* No (left), static (center), and dynamic (right) bundling.

partitions *during* search using DNPI is always beneficial: it yields larger bundles and reduces the search effort. The tree generated by dynamic bundling is shown in Fig. 5 (right). Some of the computational savings are due to the fact that the computation of DNPI with the JDT can be directly used for forward checking. The rest can be traced to bundling and factoring out no-goods. Our tests to find one solution with search combined with dynamic variable ordering [9] clearly show that, in comparison to dynamic bundling, static bundling is prohibitively expensive, particularly ineffective, and should be avoided.

The Cross Product Representation (CPR) of Hubbe and Freuder [16] yields exactly the same results as dynamic bundling with DNPI, but requires more space. It operates by doing forward checking for every value of the current variable, comparing the resulting CSPs induced on the future variables, and then bundling those values of the current variable that yield the same CSPs on the future variables. Hence, CPR necessarily visits more nodes than DNPI, even if the difference is polynomially bounded.

2.3 Phase transition

For combinatorial problems, Cheeseman et al. presented empirical evidence of the existence of a phase transition phenomenon at a critical value (cross-over point) of an order parameter as the parameter is varied. They showed that this transition results in a significant increase in the cost of solving combinatorial problems for the values around the critical value [8]. They also showed that the location of the phase transition and its steepness increase with the size of the problem. Because problems at the cross-over point are acknowledged to be probabilistically the most difficult to solve, empirical studies are typically conducted in this area to compare the performance of algorithms.

3 Bundling non-binary CSPs

No technique is reported in the literature for computing the domain partition of a CSP variable in the presence of non-binary constraints. We report here for the first time how this can be done by extending the binary case. As for the binary case, the idea is to identify, for each value in the domain of a variable the variable-value pairs in the immediate neighborhood of the variable with which it is consistent. The values that 'have the same neighborhood' form an equivalence class. The difficulty with non-binary constraints is the fact that the constraints have different arities and the 'neighborhoods' of two values are difficult to compare. Our technique is based on building a separate discrimination tree for *each* of the constraints that applies to the variable and intersects the resulting partitions. We call this tree the *non-binary discrimination tree* (NB-DT). Special care must be made to exploit the information embedded in the individual NB-DTs for forward checking. Algorithm 1 shows how to build the discrimination tree given a variable V and a constraint C that applies to it. Here σ and π correspond respectively to the selection and projection operators in relational algebra. In Fig. 6, we show the non-binary discrimination tree

```
Input: V, C
Create the root of the discrimination tree
for every value v ∈ D_V do
    for every tuple t = (⟨V_i a_i⟩, ⟨V_j a_j⟩, ..., ⟨V_k a_k⟩) ∈ C do
        if σ_{V=v}(t) then
            Move to if present, construct and move to if not, a child node in
            the tree corresponding to π_V(t)
        end
    end
    Add 'V, {v}' to the annotation of the node (or root)
    Go back to the root of the discrimination tree
end
Output: Root of discrimination tree
```

Algorithm 1: *Algorithm to create a* `NB-DT`*(V, C)*

(NB-DT) for each of the constraints incident to V in the example of Section 2.1.

After finding each of these trees, we combine the trees to find the partitioning

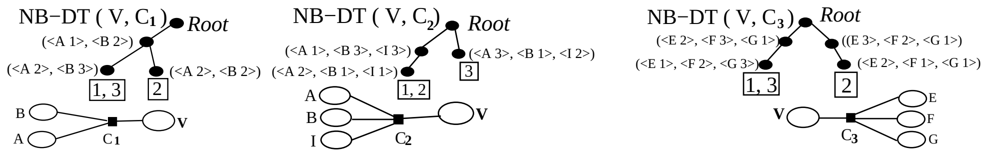

Fig. 6. *NB-DTs for the variable V and C_1, C_2, and C_3 respectively.*

of the domain of V. Combining the trees requires three sub-tasks:

1. Traverse a tree from the root to the annotations collecting, for each annotation A_i, the nodes on a path P_i in the discrimination tree leading to the annotation. Form a list $l_i = (P_i \ A_i)$ of the particular path and the corresponding annnotation. Form a list $L = \{l_i\}$ of these lists. For example, for V and C_1 in Fig. 6, , we have $l_1 = ((((\langle A\ 1\rangle, \langle B\ 2\rangle), (\langle A\ 2\rangle, \langle B\ 3\rangle)), \{1, 3\})$, $l_2 = ((((\langle A\ 1\rangle, \langle B\ 2\rangle), (\langle A\ 2\rangle, \langle B\ 2\rangle)), \{2\})$, and $L = (l_1, l_2)$.
2. Intersect the domain partitions A_i obtained from each tree. This yields the singletons $\{1\}$, $\{2\}$, and $\{3\}$ in the example of Fig. 6. The resulting sets are the equivalence classes E_i of the domain of the variable (here V) given the constraints that apply to it (here, C_1, C_2, and C_3).
3. The following step is needed to generate information to be used for forward checking. For each partition E_i, we identify the paths $\{P_i\}$ in each NB-DT such that $E_i \subseteq A_I$. For variable X connected to V we project each of the path P_i on X. Intersecting the results of the projections gives us the subset of D_X that is consistent with the values in E_i. This would be the new domain of X obtained after forward checking had we assigned E_i to V during search.

4 BT with dynamic bundling for a non-binary CSP

BT with FC using dynamic bundling operates by partitioning the domain of the current V_c using the procedure described in Section 3 and the constraints selected according to `nFC2` (Section 4.1) and updated according to Equation (2) below. The domains of the future variables are updated as described in Section 4.1. Note that we provide no guarantee that the bundling obtained by our search is maximal [17]. Further, implementing a MAC-like, full lookahead schema [22] results in better filtering of the domains of the future variables, a reduction of the number of nodes visited during search, and thus 'fatter' solution bundles at the expense of increasing the number of constraint checks. Using MAC is beneficial for sparse CSPs with tight constraints.

4.1 Non-binary FC with bundles

On binary CSPs, FC works by filtering the domains of the future variables in $\mathcal{V}_f$ connected to V_c given the instantiation of V_c. When we extend these concepts to non-binary constraints, we must decide how to deal with partially instantiated constraints with one or more future variables. We choose here to perform consistency checking on every constraint that involves both the current variable, V_c, and at least one future variable. This corresponds to the strategy known as `nFC2` [6]. In order to check a non-binary constraint, we must check every possible combination of the values remaining in the domains of the future variables. Consider the constraint C, which involves variables A, B, I, and V. Suppose that A has been instantiated to 2, leaving the domains of B, I, and V as $\{2\}$, $\{1, 3\}$, and $\{3, 4, 5, 6\}$, respectively, as shown in Fig. 7. When the search

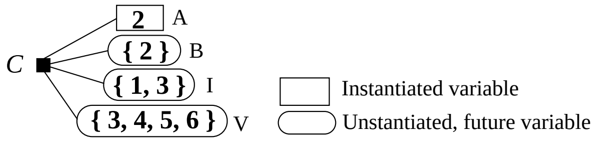

Fig. 7. *Instantiation order of variables during search.*

procedure moves to instantiate V_c, `nFC2` considers all the constraints that apply to V_c and at least one future variable $V_f \in \mathcal{V}_f$, let C_x be one such constraint. Given $\mathcal{V}_p$, the domains of the variables in $\{V_c\} \cup \mathcal{V}_f$ might have been already filtered by FC, and certain tuples in C_x might have become invalid. Thus, we need to select the tuples of C_x that have survived the filtering by FC according to $\{\langle V_i\ a_i\rangle\}$ for the variables $V_i \in \mathcal{V}_p$. We denote this operation:

$$\sigma^{FC}_{\mathcal{V}_p}(C_x). \tag{1}$$

In the example of Fig. 7, we have $V_c = B$, $\mathcal{V}_p = \{A\}$, $\mathcal{V}_f = \{I, V\}$, and $\{\langle V_i\ a_i\rangle\}$ $\{\langle A\ 2\rangle\}$. In order to compute the new domains of the variables in $\mathcal{V}_f$, we first project $\sigma^{FC}_{\mathcal{V}_p}(C_x)$ on the set $\{V_c\} \cup \mathcal{V}_p$,

$$C'_x = \pi_{\{V_c\}\cup\mathcal{V}_p}(\sigma^{FC}_{\mathcal{V}_p}(C_x)). \tag{2}$$

For a given value v in the current domain of V_c, the new domain of a future variable V_f is obtained by first making a selection on C'_x given $\langle V_c\ v\rangle$, then projecting the resulting relation on V_f:

$$D_{V_f} = \pi_{V_f}(\sigma_{\langle V_c\ v\rangle}(C'_x)). \tag{3}$$

4.2 Measuring constraints checked in non-binary FC

To assess the cost of checking constraints, we count the number of times a given vvp is compared to a tuple in a constraint. The comparisons done during non-binary FC are primarily of two types:

1. When checking whether the instantiation of a variable appears in a tuple of a constraint definition, we increment the counter by one. This type of comparison is done to select constraint tuples consistent with past instantiations.

2. When checking whether the value for a variable in a constraint tuple is present in the current domain (of size a) of that variable, we may do more than one comparison. In the worst case we will do a comparisons. This type of comparison is done to select tuples from the constraints that are consistent with domains of the current and future variables. For example let V be a future variable whose domain is $\{1, 2, 4, 3, 5\}$ (the domain is stored in the order shown). Let C_{ABV} be a constraint that applies to variables A, B, and V. Let $t = \{\langle A\ 1\rangle\ \langle B\ 2\rangle\ \langle V\ 3\rangle\}$ be a tuple from the constraint C_{ABV}. It takes four comparisons to check whether t is valid given the domain of V.

A constraint check over a k-ary constraint involves a maximum of k such checks, one for every variable of the constraint. The worst case occurs when a constraint check succeeds or fails due to the last variable of the constraint. In case of an early failure, the number of comparisons of vvps, and consequently the number of constraint checks, will be less than for the worst case. This measure accurately reflects the constraint-checking effort in our implementation. Note that we ignore the cost of making the projection, but include it in the CPU time.

5 Generating random non-binary CSPs

To analyze the performance of search algorithms and compare them, we need many instances with similar characteristics that can be controlled. Real-world problems cannot be controled by explicit parameters to enable statistical analysis. We need to generate random CSP instances by controlling the number of variables, domain size, constraint probability, constraint tightness, and constraint arity. (Typically, domain size and constraint tightness are uniform throughout the generated instance.) Few generators exist that allow one to control the arity of the constraints [5, 23]. Moreover, we have not come across any generator that guarantees the existence of a solution. Below, we describe such a generator.

Constraint probabilty in non-binary CSPs: Figs. 1 and 2 show that the scope of constraints may overlap. Here, scope$(C_1) \subset$ scope(C_2). A non-binary CSP can have an arbitrary number of such overlapping constraints, so it is difficult to define the overall constraint probability of the CSP. Let p_k indicate the constraint probability of all constraints of arity k, $p_k = \frac{c_k}{\binom{n}{k}}$, where c_k is the number of k-ary constraints in the CSP.

Non-binary random CSP generator: Our generator takes as input the following parameters: number of variables (n), domain size (a), constraint tightness (t), and constraint probability p_k for each constraint arity k to be generated. Constraint tightness t is defined as the ratio of the number of forbidden tuples over that of possible tuples, which is a^k for a k-ary constraint. First, we generate a solution by randomly assigning values to each variable in the CSP. We compute c_k for each k in $\{2, 3, \ldots n\}$. We then assign variables to constraints and check if the CSP is connected. If it is not connected we try another assignment of variables to constraints until we get a connected CSP. We project, on all the constraints, the solution initially generated thereby ensuring that each constraint has the tuple necessary for the global solution. We then randomly set ta^k tuples to be disallowed, making sure that the tuples corresponding to the solution are not eliminated. Thus, we are able to generate random non-binary connected CSPs with a specified constraint tightness and probability that are guaranteed to have at least one solution.

6 Experiments

The benchmark problems usually used for symmetric CSPs are not suitable for testing the performance of bundling for the following reasons:

– Most of these problems exhibit only symmetries in terms of permutation (i.e., isomorphic interchangeability, which is orthogonal to domain bunding).
– Most of these problems (e.g., game of life, balanced incomplete block design) have small domains (e.g., binary), which are not amenable to bundling.
– Most of these problems are modeled using a unique global constraint of exponential size (e.g., game of life). Defining the constraint in extension amounts to solving the problem.
– Finally, in coloring problems symmetries are typically permutations. Domain bundling is beneficial in the case of list-coloring problems and common in resource-allocation applications and can be easily computed without requiring NB-DTs as shown in [10].

We compared the performance of search with dynamic bundling (DNPI-FC-`DDD`) and with no bundling (FC-`DDD`) on toy problems (e.g., Hoffman-Clowes scene labeling and Zebra). While dynamic bundling could not bundle up the solutions of the toy problems, it reduced both the number of constraint checks and CPU time. This is explained by the fact that DNPI bundles no-goods. We also ran experiments on problems generated randomly using our generator. We generated six sets with the same parameters as the ones described in [6], except for the fourth set. The problems are identified by the tuple $\langle k, n, a, p_k, t\rangle$ as follows.

1. $\langle 3, 10, 10, 0.8300, t\rangle$: High constraint probability.
2. $\langle 3, 30, 6, 0.0180, t\rangle$: Moderate constraint probability.
3. $\langle 3, 75, 5, 0.0018, t\rangle$: Sparse constraint probability.
4. $\langle 4, 10, 8, 10^{-1}, t\rangle$: High constraint probability.
5. $\langle 4, 26, 6, 10^{-2.5}, t\rangle$: Moderate constraint probability.
6. $\langle 4, 63, 4, 10^{-4}, t\rangle$: Sparse constraint probability.

To find the first solution, we varied tightness t in [0.05, 0.10, ..., 0.95] with a step of 0.025 around the cross-over point and generated 30 instances for each case. We averaged the values of number of nodes visited `NV`, number of constraint checks `CC`, the first bundle size `FBS`, and CPU time. We show the results in Figs. 8 and 9. Tab. 1 shows the results of experiments on three sets of data with 4-arity constraints at the cross-over point. The results found in the region of phase

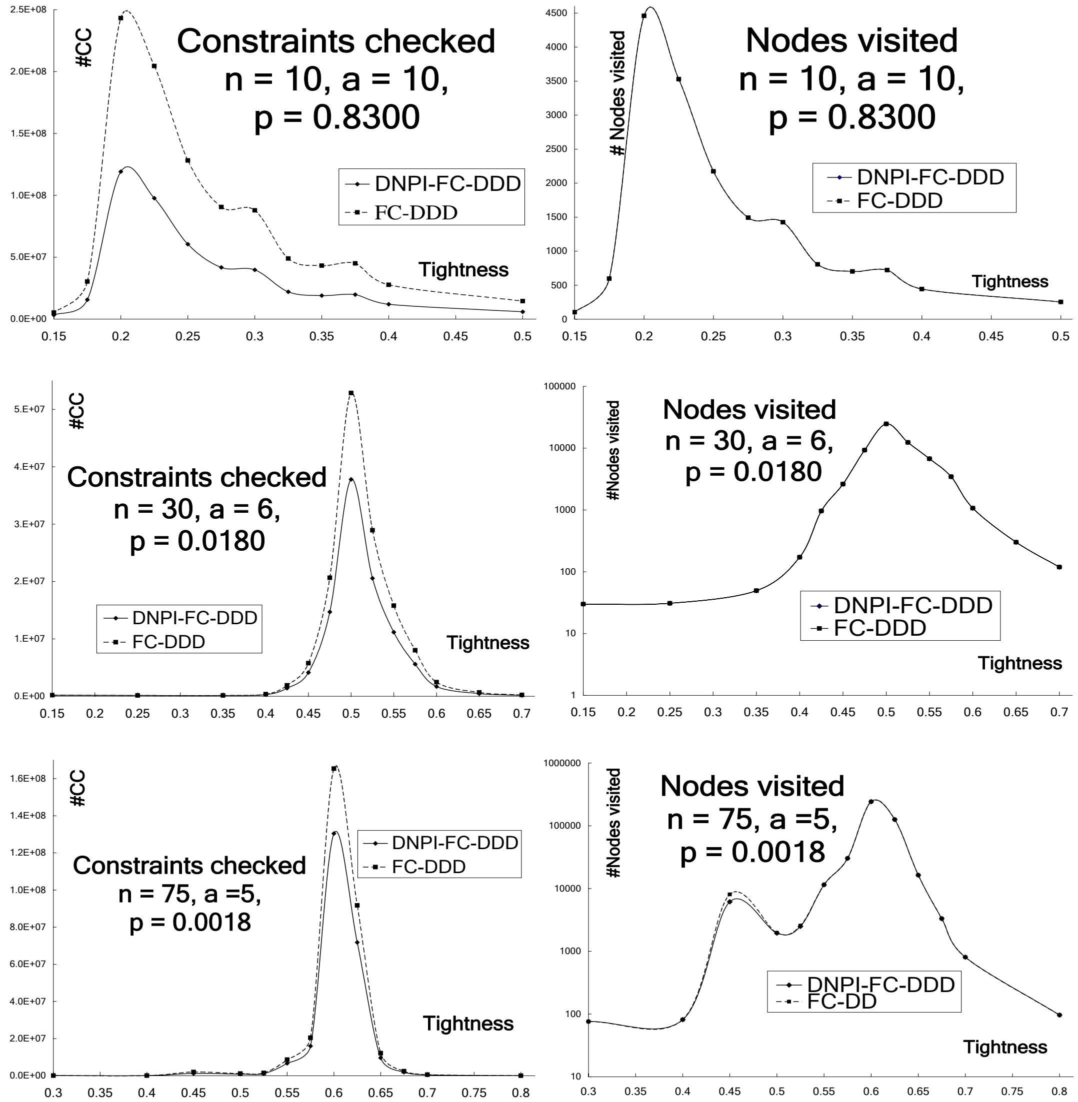

Fig. 8. *Constraint checks and nodes visited for finding one solution for the data sets with ternary constraints.*

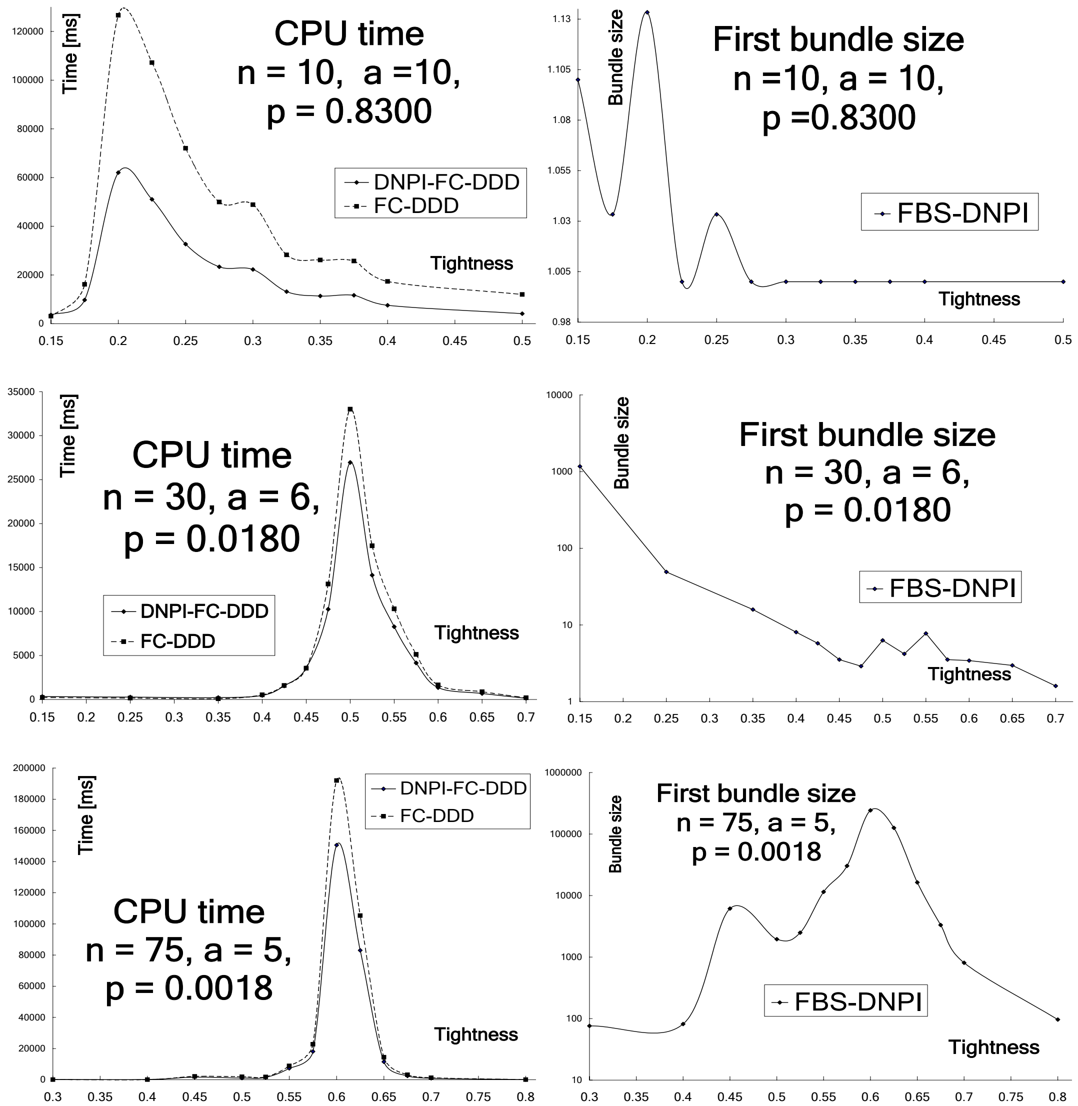

Fig. 9. *CPU time and first bundle size for finding one solution for the data sets with ternary constraints.*

	n	a	p_4	t	#CC	FBS	#NV	CPU time (secs)
FC-DDD	10	8	10^{-1}	0.650	$126.56{\cdot}10^6$	1	4322.7	84.13
DNPI-FC-DDD	10	8	10^{-1}	0.650	$70.25{\cdot}10^6$	1.1	4322.7	47.98
FC-DDD	26	6	$10^{-2.5}$	0.625	$736.56{\cdot}10^6$	1	201249.4	1094.16
DNPI-FC-DDD	26	6	$10^{-2.5}$	0.625	$408.86{\cdot}10^6$	1.86	201249.3	801.66
FC-DDD	63	4	10^{-4}	0.750	$39.07{\cdot}10^6$	1	44433.4	45.8
DNPI-FC-DDD	63	4	10^{-4}	0.750	$30.77{\cdot}10^6$	68.9	44432.3	34.91

Table 1. *Random problems with constraint arity 4 around the cross-over point.*

transition hold uniformly throughout the spectrum, unless specified. To find all solutions, we ran similar tests on smaller problems (i.e., $n = 10$ and $a = 5$).

Observations and analysis. As for binary CSPs, bundling does amazingly well for finding all solutions and also the first solution. Once again, dynamic bundling outperforms non-bundling. In particular, DNPI-FC-DDD never visits more nodes and never checks more constraints than than FC-DDD. Further, DNPI-FC-DDD always takes significantly less CPU time to find all solutions than FC-DDD, up to an order of magnitude.

Observation 1 *Finding the first solution in non-binary CSPs.* DNPI-FC-DDD never visits more nodes or performs more constraint checks than FC-DDD. CPU time is also generally better except when DNPI-FC-DDD finds large solution bundles, which usually occurs in low-tightness regions. Even then CPU time is never significantly higher for dynamic bundling than for no-bundling.

Observation 2 *Phase transitions in non-binary CSPs.* DNPI-FC-DDD systematically performs better than FC-DDD in the phase transition region.

We also tested other variable ordering heuristics such as Static Least Domain (SLD) and Dynamic Least Domain (DLD). We found that, for our set of random problems, SLD was not effective. DLD was better than SLD. However, the best results were obtained with DDD. We do not claim that DDD is always better than DLD since, in our experience with real-world problems, there is no clear winner.

7 Conclusions and future work

Non-binary constraints are important to model faithfully the constraints of many real-world problems. In this paper, we described a technique for bundling dynamically the search space of non-binary CSPs. We demonstrate that in non-binary CSPs, dynamic bundling is capable of bundling the solution space of CSPs and solving them faster than a non-bundling search procedure. The improvement is particularly significant in the region of the cross-over point. This result is a particularly interesting one, heavy with promise for new applications such as design and relational databases. We are working on applying the compact solution space generated by dynamic bundling to problems of query optimization using materialized views in databases.

Acknowledgments. Preliminary results were reported in [11]. This work is supported by the Maude Hammond Fling Faculty Research Fellowship and CAREER Award #0133568 from the National Science Foundation. Deb Derrick provided editorial help.

References

1. F. Bacchus and P. van Beek. On the Conversion between Non-Binary and Binary Constraint Satisfaction Problems Using the Hidden Variable Method. *AAAI-98*.
2. A.M. Beckwith and B.Y. Choueiry. On the Dynamic Detection of Interchangeability in Finite Constraint Satisfaction Problems. *CP 2001*.
3. A.M. Beckwith, B.Y. Choueiry, and H. Zou. How the Level of Interchangeability Embedded in a Finite Constraint Satisfaction Problem Affects the Performance of Search. *Australian Joint Conference on Artificial Intelligence. LNAI 2256*, 2001.
4. B.W. Benson and E.C. Freuder. Interchangeability Preprocessing Can Improve Forward Checking Search. In *Proc. of the 10 th ECAI*, pages 28–30, 1992.
5. C. Bessière. A Generator for Non-Binary CSPs. Email communication, 2001.
6. C. Bessière, P. Meseguer, E.C. Freuder, and J. Larrosa. On Forward Checking for Non-binary Constraint Satisfaction. *Artificial Intelligence*, 141:205–224, 2002.
7. C. Bessière, P. Meseguer, E.C. Freuder, and J.-Ch. Régin. On Forward Checking for Non-binary Constraint Satisfaction. *CP'99. LNAI 1330*, 1999.
8. P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the Really Hard Problems Are. *IJCAI-91*, pages 331–337, 1991.
9. B.Y. Choueiry and A.M. Davis. Dynamic Bundling: Less Effort for More Solutions. *Symp. on Abstraction, Reformulation & Approximation*, LNAI 2371, 2002.
10. B. Y. Choueiry and G. Noubir. On the Computation of Local Interchangeability in Discrete Constraint Satisfaction Problems. *AAAI-98*, pages 326–333, 1998.
11. A. Davis. Dynamically Detecting and Exploiting Symmetry in Finite Constraint Satisfaction Problems. Master's thesis, CSE-UNL, April 2002.
12. T. Ellman. Abstraction via Approximate Symmetry. *IJCAI-93*.
13. E.C. Freuder. Eliminating Interchangeable Values in Constraint Satisfaction Problems. *AAAI-91*, pages 227–233, 1991.
14. J.W.L. Glaisher. On the Problem of the Eight Queens. *Philosophical Magazine, series 4*, 48:457–467, 1874.
15. A. Haselböck. Exploiting Interchangeabilities in Constraint Satisfaction Problems. *IJCAI-93*, pages 282–287, 1993.
16. P.D. Hubbe and E.C. Freuder. An Efficient Cross Product Representation of the Constraint Satisfaction Problem Search Space. *AAAI-92*, pages 421–427, 1992.
17. D. Lesaint. Maximal Sets of Solutions for Constraint Satisfaction Problems. *ECAI-94*, pages 110–114, 1994.
18. N. Neagu and B. Faltings. Exploiting Interchangeabilities for Case Adaptation. In *International Conference on Case-Based Reasoning (ICCBR 01)*, 2001.
19. A. Petcu and B. Faltings. Applying Interchangeability Techniques to the Distributed Breakout Algorithm. *IJCAI 2003*, pages 1374–1375,2003.
20. J.-F. Puget. On the Satisfiability of Symmetrical Constrained Satisfaction Problems. In *ISMIS'93*, pages 350–361, 1993.
21. F. Rossi, Ch. Petrie, and V. Dhar. On the Equivalence of Constraint Satisfaction Problems. *ECAI-90*, pages 550–556, 1990.
22. D. Sabin and E.C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. *ECAI-94*, pages 125–129, 1994.
23. H. Zou, B.Y. Choueiry, and A.M. Davis. A Generator of Random Non-Binary Finite Constraint Satisfaction Problems. TR CONSYSTLAB-02-02, UNL, 2002.