# Neighborhood Interchangeability and Dynamic Bundling for Non-Binary Finite CSPs

Anagh Lal[1], Berthe Y. Choueiry[1], and Eugene C. Freuder[2]

[1] Constraint Systems Laboratory
Computer Science & Engineering
University of Nebraska-Lincoln
alal,choueiry@cse.unl.edu
[2] Cork Constraint Computation Center
Department of Computer Science
University College Cork, Ireland
e.freuder@4c.ucc.ie

**Abstract.** Neighborhood Interchangeability (NI) identifies the equivalent values in the domain of a variable in a Constraint Satisfaction Problem (CSP) by considering only the constraints that directly apply to the variable. Freuder described how to compute the NI values using the discrimination tree (DT) [1]. The generalization of DT to non-binary CSPs is not straightforward because the constraints that apply to a given variable have different scopes and arities. In this paper, we introduce a method[3] for computing the NI values in the presence of non-binary constraints. Then, we demonstrate the advantages of computing these NI values dynamically during backtrack search, in a process we call dynamic bundling. As for the binary case [2, 3], we show that in addition to yielding a robust solutions and a compact representation of the solution space, dynamic bundling can significantly improve the performance of the search process (which is a fortunate and unexpected side-effect). On randomly generated problems, the performance improvement is particularly significant around the cross-over point. In [4], we discuss the advantages of applying these techniques to improve join computation in databases.

## 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 [5], Puget [6], Ellman [7] and many others proposed exploiting *declared* symmetries specific to a class of problems to improve the performance of search. In addition to *exact* symmetries, Ellman also considered necessary and sufficient *approximations* of symmetry relations. While the above approaches focused on exploiting *declared* symmetry, our study considers the *discovery* and use of symmetries. The symmetry mechanisms we study are based on the notions of local value interchangeability of Freuder [1] and domain bundling of Haselböck [8], which groups equivalent values in a bundle (or equivalence

[3] The algorithms and their application to databases are the subject of a pending patent.

class). It was incorrectly assumed that bundling, applied statically (i.e., prior to search) and, a fortiriori, dynamically (i.e., during search), is too costly and not worthwhile for finding a single solution. Beckwith et al. [2] and Choueiry and Davis [3] showed how to implement bundling to yield multiple and robust solutions for less effort than needed for finding a single solution. (This result holds theoretically for finding all solutions, and empirically for finding one solution.) They also showed that dynamic bundling is significantly less expensive and more effective than static bundling. However, their techniques were limited to binary CSPs.

Although most research in constraint satisfaction focuses on binary CSPs, many real-life problems are more 'naturally' modeled as non-binary CSPs. The focus on binary CSPs has so far been tolerated because it is always possible in principle to reduce a finite non-binary CSP to a binary one [9, 10]. Research on non-binary constraints is still in its infancy and the traditional attitudes on this issue are now being challenged [11]: it appeared that sometimes it is more effective to operate on the non-binary encoding of the CSP than on its binary reduction.

In this paper we introduce an efficient technique for bundling non-binary CSPs and demonstrate its effectiveness on toy and randomly-generated problems. Even though such problems lack the redundancy one expects to find in real-world applications, making them not particularly amenable to bundling, our empirical results show that bundling remains beneficial. Our contributions are:

1. An algorithm for computing the NI values of a CSP variable given any subset of the constraints that apply to the variable *regardless of their arities*.
2. The integration of this mechanism with backtrack search, which we call dynamic bundling, for solving non-binary CSPs.
3. Extensive experiments that demonstrate the benefits of dynamic bundling.

This paper is organized as follows. Section 2 states the motivations and background of our work. Section 3 shows how to compute NI values in the presence of non-binary constraints and to integrate bundling with search using non-binary forward-checking. It also illustrates with an example how solutions and no-goods are bundled. Section 4 reports our experiments and analysis. Finally, Section 5 concludes this paper and gives directions for future research.

## 2 Motivation and Background

Beckwith et al. studied dynamic bundling in the context of binary CSPs. They have established that dynamic bundling is guaranteed to never cost more than no-bundling when seeking all solutions. In particular, they established that, for 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 [2]. Choueiry and Davis showed empirically that these results hold when seeking the first solution [3]. Those results prove 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. This counter-intuitive result is explained by the fact that *dynamic bundling is capable of bundling no-goods*, defined as partial solutions that cannot yield complete solutions. Thus, dynamic bundling appears as a double-edged sword that reduces thrashing during search. Our goals here are to extend neighborhood interchangeability to non-binary CSPs and to establish the benefits of dynamic bundling, especially in the region of the phase transition. We restrict ourselves in this paper to presenting the methods and evaluating them on randomly generated problems. However, we have already shown their advantages in the context of databases [4]. Further, we believe that the extension to non-binary case will be more useful than the binary one, which proved beneficial in case-based reasoning [12] and local search [13].

### 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. 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 $V_i$, $V_j$, $\ldots$, $V_k$ is specified as a set of tuples, subset of the cross-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 assume that the domains of the variables are finite. Solving a CSP requires assigning a value to each variable such that all constraints are simultaneously satisfied. The problem is **NP**-complete in general. A CSP is often represented as a graph, or constraint network. In this graph, 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, as shown in Figure 1. The constraint definitions for this example are given in Figure 2. We use the

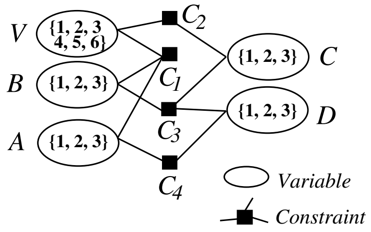


**Fig. 1.** *Example of a non-binary CSP.*

| $C_1$ | | |
|---|---|---|
| $V$ | $A$ | $B$ |
| 1 | 1 | 3 |
| 1 | 3 | 3 |
| 2 | 1 | 3 |
| 2 | 3 | 3 |
| 3 | 1 | 1 |
| 3 | 2 | 2 |
| 4 | 1 | 1 |
| 4 | 2 | 2 |
| 5 | 3 | 2 |
| 6 | 3 | 2 |

| $C_2$ | |
|---|---|
| $V$ | $C$ |
| 1 | 3 |
| 2 | 3 |
| 3 | 2 |
| 4 | 2 |
| 6 | 1 |

| $C_3$ | | |
|---|---|---|
| $B$ | $C$ | $D$ |
| 1 | 2 | 1 |
| 1 | 2 | 2 |
| 2 | 2 | 1 |
| 2 | 2 | 2 |
| 3 | 1 | 1 |

| $C_4$ | |
|---|---|
| $A$ | $D$ |
| 1 | 1 |
| 2 | 2 |
| 3 | 1 |

**Fig. 2.** *Constraint tables.*

following parameters to assess the worst-case complexity of an algorithm applied to a CSP and for generating random instances: $n$ number of variables, $a$ maximum domain size, $d$ node degree, $c_k$ number of constraints of arity $k$, $p_k = \frac{c_k}{\binom{n}{k}}$ constraint probability 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 (BT). In this paper, we use forward checking (FC) and order the variables dynamically during search according to the least domain heuristic. Depth-first search for binary CSPs proceeds by iteratively choosing a current variable $V_c$ and instantiating it, i.e. assigning to it a value taken from its domain. The process repeats until all variables are instantiated. Uninstantiated variables are called future variables, and their set is denoted by $\mathcal{V}_f$. FC propagates the effect of instantiating the current variable by removing values inconsistent with $a$ from the domains of the future variables adjacent to $V_c$. If the instantiation does not wipe out the domain of any variable in $\mathcal{V}_f$, $V_c$ is added to the set of instantiated variables (which we call past variables and denote as $\mathcal{V}_p$) and search proceeds to the next variable determined by the ordering schema used. 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 assignment, and revokes the assignment done at this level. 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 instantiations $a_i$. Search on non-binary CSPs proceeds as described above but FC requires particular attention as discussed in Section 3.3. We call a *no-good* any combination of variable-value pairs that cannot be extended to a consistent solution.

### 2.2 Interchangeability

Interchangeability is about the ability to recover one solution to a CSP from another. 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. In the broadest sense, this is *functional* interchangeability [1]. We address here a restricted form of interchangeability: the interchangeability of values in the domain of a single variable. This type of interchangeability does not cover the permutation of values across variables, which is an isomorphic interchangeability. Below we recall some forms of interchangeability relevant to our work.

**Definition 1.** Full interchangeability (FI) (Freuder [1]): *Values $a$, $b \in D_V$ are FI iff every CSP solution involving $a$ remains a solution when $b$ is substituted for $a$, and vice versa.*

Checking all the solutions of the CSP in Figure 3 we find that the values $d$, $e$, and $f$ are fully interchangeable for $V_2$. Computing full interchangeability may require finding

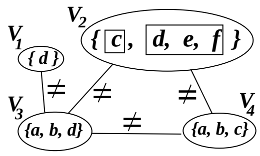


**Fig. 3.** *A binary CSP.*

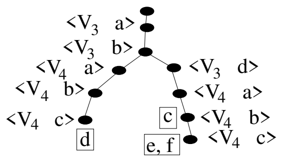


**Fig. 4.** *Partitioning $D_{V_2}$.*

all solutions and hence is likely to be intractable. Freuder [1] identified a form of *lo-*

*cal* interchangeability, called *neighborhood interchangeability* (NI), that is a sufficient approximation of full interchangeability.

**Definition 2.** Neighborhood interchangeability (NI) (Freuder [1]): *A value $a \in D_V$ is neighborhood interchangeable with a value $b \in D_V$ 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 $C\}$.

NI is a sufficient, but not a necessary condition for FI. Indeed, in the CSP of Figure 3, only values $e$ and $f$ are NI for $V_2$ whereas values $e$, $f$, and $d$ are FI for $V_2$. Algorithm 1, introduced in [1], computes the NI values for a variable $V$ by building a discrimination tree (DT).

```
Input: V
current-node ← Create the root of the discrimination tree for each value v ∈ D_V do
    for each variable V_j ∈ NEIGHBORS(V) do
        for each value w ∈ D_{V_j} consistent with v for V do
            if current-node has a child node n with '⟨V_j w⟩' then
                current-node ← n
            end
            else
                Generate n a node with '⟨V_j w⟩' and make it a child of current-node
                current-node ← n
            end
        end
        Add 'V, {v}' to annotation of current-node
        current-node ← root of the discrimination tree
    end
end
Output: Root of discrimination tree
```

**Algorithm 1:** *Algorithm to create a DT of a variable $V$.*

Figure 4 shows the discrimination tree generated for $V_2$ of the CSP of Figure 3. In this tree, the nodes represent variable-value pairs in the neighborhood of $V_2$. Further, 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. The complexity of this procedure is $O(n \cdot a^2)$. It is important, in this procedure, that variables and values be ordered in a canonical way.

Benson and Freuder used NI to improve search [14]. A weaker form of NI, called *neighborhood interchangeability according to one constraint* ($\mathrm{NI}_C$), was also used in search by Haselböck [8]. 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. We call such strategies *static bundling*. Figure 5 shows a search tree for the example of Figure 3 without bundling (left) and with static bundling (center).

Freuder [1] noticed that computing interchangeability *during* problem solving results

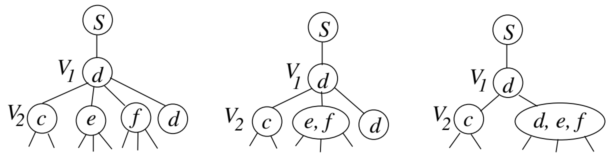


**Fig. 5.** *Search with no, static, and dynamic bundling.*

in a weak type of interchangeability, *dynamic interchangeability*. Beckwith et al. [2] and Choueiry and Davis [3] established that recomputing interchangeability partitions *during* search is always beneficial: it yields larger bundles and reduces the search effort. We call the strategy *dynamic bundling* (DynBndl). Figure 5 (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 FC. Further, they showed that, in comparison to dynamic bundling, static bundling is prohibitively expensive, particularly ineffective, and should be avoided [3].

The Cross Product Representation (CPR) of Hubbe and Freuder [15] yields the same resulting bundles as dynamic bundling, but it requires more space and does not bundle no-goods. It operates by doing forward checking for every value of the current variable, comparing the CSPs induced on the future variables, and then bundling the values of the current variable yielding the same induced CSPs. Hence, CPR necessarily visits more nodes than DynBndl, even though the difference is polynomially bounded.

### 2.3 Phase transition

Cheeseman et al. [16] presented empirical evidence, for some random combinatorial problems, of the existence of a phase transition phenomenon at a critical value (cross-over point) of an order parameter. They showed a significant increase in the cost of solving these problems around the critical value. They also showed that the location of the phase transition and its steepness change 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 to compare the performance of algorithms are typically conducted in this area.

## 3 Dynamic Bundling for non-binary CSPs

In this section we first describe how to compute 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 Neighborhood interchangeability for non-binary constraints

No technique is reported in the literature for bundling non-binary CSPs. Here we report for the first time how this can be done by extending the binary case. The idea is to identify the variable-value pairs in the neighborhood of a variable $V$ consistent with each value in $D_V$. The values with 'the same neighborhood' form an equivalence class. The difficulty with non-binary constraints is that the constraints have different arities and the 'neighborhoods' of two values are difficult to compare. This makes the transition from binary to non-binary CSPs non-trivial because one discrimination tree cannot represent tuples from all the constraints involved. Our technique is based on building a separate discrimination tree for *each* of the $d$ constraints that apply to the variable. We call such a tree a *non-binary discrimination tree* (nb-DT). Below, we introduce two processes. The first partitions the domain of the variable by building and combining the applicable nb-DTs; and the second determines the domains of the neighboring variables consistent with each set of the partition. These two processes are used for dynamic bundling (Section 3.3).

**Finding the domain partition.** The first process uses Algorithm 2 to build the nb-DT for a variable $V$ and an associated constraint $C$ that applies to it. Here $\sigma$ and $\pi$ correspond respectively to the selection and projection operators in relational algebra. An nb-DT is created for each of the $d$ constraints applying on $V$.

```
Input: V, C
current-node ← Create the root of the discrimination tree
S ← scope(C) \ {V}
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
            if current-node has a child node n with π_S(t) then
                current-node ← n
            end
            else
                Generate n a node with π_S(t) and make it a child of current-node
                current-node ← n
            end
        end
    end
    Add 'V, {v}' to the annotation of current-node
    current-node ← root of the discrimination tree
end
Output: Root of discrimination tree
```

**Algorithm 2:** *Algorithm to create a nb-DT ($V$, $C$)*

Figure 6 shows the non-binary discrimination tree (nb-DT) for the constraints incident to $V$ in the example of Figures 1 and 2. The worst-case time complexity algorithm

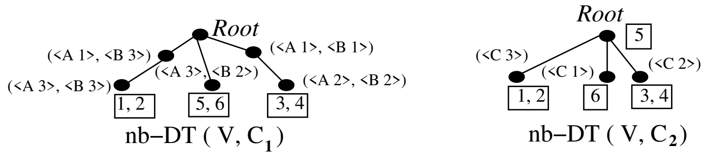


**Fig. 6.** *nb-DTs for $V$ and $C_1$, $C_2$, respectively.*

is linear in the size of the constraint, which depends on the domain size of the variable, the tightness, and the arity of the constraint. The complexity for building $d$ such nb-DTs is therefore $O(d \cdot a^{k+1} \cdot (1-t))$. Every node of the nb-DT stores the tuple it represents, a list of nodes connected to it, and an annotation which is by default empty. A pointer called the *current-node* is maintained and it points to the last node used by the algorithm. Initially, the *current-node* points to the *Root*. The algorithm builds the tree choosing one value $v$ from $D_V$ and processing each tuple of $C$ corresponding to $\langle V\ v\rangle$ as follows: When the projected tuple matches any of the child nodes of the *current-node*, the *current-node* moves to the matching node. Otherwise, a new node is created and added to the *current-node*'s children list and the *current-node* moves to the newly created node. After processing a $v \in D_V$, $v$ is added to the annotation of the *current-node* and the *current-node* moves back to the *Root*. Therefore, two nodes are connected if the tuples of each of these nodes lie on the path to a common annotation value. The trees generated for each of the constraints are then combined as follows to determine the partition of $D_V$:

1. Traverse each tree from the root to each annotation $A_i$ and construct $P_i$ by collecting the nodes on the path. 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. For the example of Figure 6, we have:
   (a) For the nb-DT of $C_1$:
      - $l_1$ = $(((\langle A\ 1\rangle, \langle B\ 3\rangle), (\langle A\ 3\rangle, \langle B\ 3\rangle)), \{1,2\})$,
      - $l_2$ = $(((\langle A\ 3\rangle, \langle B\ 2\rangle)), \{5,6\})$,
      - $l_3$ = $(((\langle A\ 1\rangle, \langle B\ 1\rangle), (\langle A\ 2\rangle, \langle B\ 2\rangle)), \{3,4\})$.
      - Thus, $L_1 = (l_1, l_2, l_3)$
   (b) For the nb-DT of $C_2$:
      - $l_4$ = $(((\langle C\ 3\rangle)), \{1,2\})$,
      - $l_5$ = $(((\langle C\ \texttt{nil}\rangle)), \{5\})$,
      - $l_6$ = $(((\langle C\ 2\rangle)), \{3,4\})$,
      - $l_7$ = $(((\langle C\ 2\rangle)), \{6\})$, Thus, $L_2 = (l_4, l_5, l_6, l_7)$.

   These lists are collected in $L = (L_1, L_2, \ldots, L_d)$.
2. Apply Algorithm 3 to $L$ and $V$, as input parameters, to intersect the annotations $A_i$ obtained from each tree in order to compute the domain partition of $D_V$. The worst-case time complexity of this algorithm is $O(d^2 \cdot a^4)$. For the example of Figure 6, the algorithm returns $\{\{1,2\},\{3,4\},\{5\},\{6\}\}$ as the partitioned domain of $V$. The resulting sets are the equivalence classes $E_i$ of the domain of the variable (here $D_V$) given the constraints that apply to it (here, $C_1$ and $C_2$).

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

**Algorithm 3:** *Algorithm to intersect annotations.*

**Computing the new domains of neighboring variables.** The second process exploits these trees to determine the values of the neighborhood of $V$ consistent with each partition of $D_V$. For a given partition $E_i$, we identify the paths $\{P_i\}$ in each nb-DT such that $E_i \subseteq A_i$. For a variable $X$ connected to $V$, we project each 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.)

**Avenues for improving performance.** In an effort to reduce the overhead for computing bundles, we have included in our implementation a mechanism for automatically 'switching off' some operations for partitioning the domain of a given variable $V$ when it becomes clear that all partitions are necessarily singletons. This happens in two situations. When any nb-DT of a $V$ results in annotations exclusively made of singleton elements (see Algorithm 2). In this case we can safely switch off bundling for building the nb-DTs for the remaining constraints that apply to the variable $V$. Another case is when the intersection of the annotations returns singletons (see Algorithm 3). In practice, we implement this switching off mechanism as follows. We force Algorithm 2 not to check for matching child nodes, but to always create new nodes because the information in the nb-DTs is useful for filtering the domains of the future variables. Another avenue for improvement is by sorting the constraint definitions. In [4] we showed a space efficient, sort-based bundling strategy applicable to database join queries. We are currently exploring ways to reduce the complexity of Algorithms 2 and 3 by exploring sort-based algorithms suitable for CSPs. Sort-based methods have better time and space complexity than Algorithms 2 and 3, but they require the constraints to be sorted. When a dynamic variable ordering is used for solving a CSP, sort-based methods require frequent re-sorting. There is thus a trade-off to be made from the benefit of sort-based methods in conjunction with dynamic variable ordering in search for solving CSPs, which needs to be further investigated.

### 3.2 Updating non-binary constraints for forward checking

Independently of bundling, two issues arise when applying FC to non-binary CSPs: (1) choosing the subset of constraints to take into account, and (2) updating their definitions to reflect past instantiations and domain prunings. We adopt the strategy called nFC2 [11], where the constraints considered are the ones that apply to the current variable and at least one future variable.

The update of a non-binary constraint according to past instantiations amounts to intersecting the original definition of the constraint with the cross product of the (updated) domains of $V_c$ and future variables. This operation is time consuming in practice. We propose here an equivalent, more efficient implementation that uses a linear number of selection and projection operations. Let $V_c$ be the current variable and $C_x$ one such constraint (see Figure 7). Let scope($C_x$) = $\{V_a\} \cup \{V_c\} \cup \{V_b\}$, where $\{V_a\} \subset \mathcal{V}_p$ and

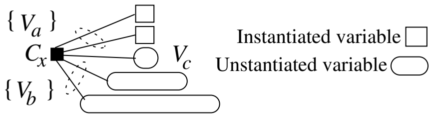


**Fig. 7.** *Partially instantiated non-binary constraint $C_x$.*

$\{V_b\} \subset \mathcal{V}_f$. The domains of variables in $\{V_c\} \cup \{V_b\}$ might have already been 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 instantiations of the past variables. The selected tuples must satisfy the conditions: (1) $\langle V_i\ a_i\rangle$ for $V_i \in \{V_a\}$ and $a_i$ the bundle instantiated to $V_i$, and (2) $a_j \in D_{V_j}$ for $V_j \in \{V_c\} \cup \{V_a\}$, where $D_{V_j}$ are filtered domains. We denote this operation $\sigma^{FC}_{\mathcal{V}_p}(C_x)$. In order to compute the updated constraint, we project $\sigma^{FC}_{\mathcal{V}_p}(C_x)$ on $\{V_c\} \cup \{V_b\}$,

$$C'_x = \pi_{\{V_c\}\cup\{V_b\}}(\sigma^{FC}_{\mathcal{V}_p}(C_x)). \tag{1}$$

The way non-binary FC without bundling is implemented affects, to a large extent, the number of constraint checks and CPU time spent to solve a CSP. The updated constraint of Equation (1) is valid for all values in $D_{V_c}$. It is wasteful to discard the result of this computation after instantiating $V_c$. If the instantiation is not consistent and the search backtracks to the variable then $C'_x$ is computed again. To avoid this expensive computation we store each $C'_x$ associated with $V_c$. Note that by doing so we level the playing field for the two algorithms being compared. Thus, our empirical results reflect the gain due purely to bundling and exclude the gains from the additional nb-DT data structure.

### 3.3 Dynamic bundling

Search using dynamic bundling 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 the first process of Section 3.1 (i.e., finding the domain partition) to the subset of the constraints on $V_c$ determined according to nFC2. The constraints passed to Algorithm 2 are computed using Equation (1). The effects of this instantiation are then propagated using the second process of Section 3.1 (i.e., computing the new domains of neighboring variables).

Figure 8 shows partially the non-bundled search of tree of the example of Figures 1 and 2 with variable ordering $\{V, A, B, C, D\}$. Figure 9 shows the dynamically bundled search tree. The domain of $V$ is partitioned as discussed in Section 3.1 and $V$ is

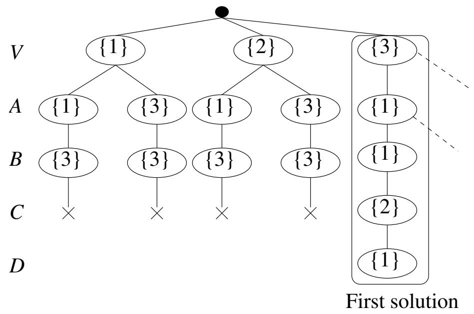


**Fig. 8.** *Search tree without bundling.*

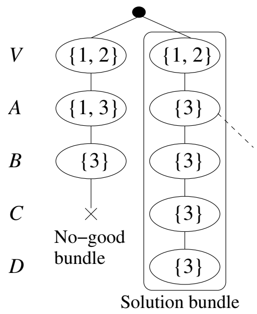


**Fig. 9.** *Search tree using dynamic bundling.*

assigned the bundle $\{1, 2\}$. FC propogates this instantiation and the domains of $A$, $B$, and $C$ are set to $\{\{1\}, \{3\}\}$, $\{3\}$, and $\{3\}$, respectively. Next, the domain partitions of $A$ are computed. We find the two domain values, 1 and 3, to be interchangeable and $A$ is instantiated with $\{1, 3\}$. On propagating this instantiation, the domain of $D$ now becomes $\{1\}$. Next, the search proceeds to instantiate $B$ with the only domain value $\{3\}$. The instantiation results in the annihalation of the domain of $C$, and search backtracks. There are no values remaining in the domains of $B$ and $A$. Hence, the search backtracks to $V$. Note here that $A$ was assigned a bundle of size 2. In the case of non-bundling search, another value from $A$ will be tested and extra nodes visited only to detect failure. By clubbing $\{1, 3\}$ of $A$ together, bundling saved visiting nodes of $B$. Further savings result because bundling detects that the bundle $\{1, 2\}$ is a also no-good. The gains due to bundling $V$ multiply those due to bundling of $A$, thus illustrating the gains of no-good bundling.

On instantiating $V$ with $\{3, 4\}$, search is able to assign values to the remaining variables of the CSP and finds a solution as $\{\langle V\ \{3, 4\}\rangle, \langle A\ \{1\}\rangle, \langle B\ \{1\}\rangle, \langle C\ \{2\}\rangle, \langle D\ \{1\}\rangle\}$. From this example, we see that dynamic bundling necessarily visits less nodes by bundling solutions and no-goods. Search with bundling visits 8 nodes whereas search without bundling will visit 15 nodes to find a solution. Moreover, bundling provides two solutions instead of one at a lower cost.

The use of a MAC-like, full lookahead schema [17] necessarily performs a better filtering of the domains of the future variables. While this may increase the number of constraint checks, it would yield 'fatter' solution bundles (thus improving bundling), and reduce of number of nodes visited during search. However, even with MAC, our technique does not guarantee that the resulting bundling is maximal [18]. More generally, dynamic bundling, while it partitions the set of solutions (i.e., every solution appears in exactly one bundle) does not guarantee the quality of the partition, its optimality, or its uniqueness.

## 4 Experiments

The benchmark problems usually used for symmetric CSPs are not suitable for bundling for the following reasons. (1) Most exhibit only symmetries that are permutations of values over variables. (2) Most have small domains (e.g., Boolean), which are not amenable to bundling. (3) Most are modeled using a unique global constraint of exponential size. Defining the constraint in extension amounts to solving the problem and is likely intractable. (4) Finally, for coloring problems, bundling can be done only in the case of list-coloring problems (typically used to model resource allocation problems). However, such bundling can be easily computed without nb-DTs as shown in [19].

For finding all solutions and using the same look-ahead strategy[4] during search, dynamic bundling (DynBndl) is guaranteed to visit no more nodes and do no more constraint checks than backtrack search without bundling. However, the bundling effort may increase the CPU time, which could be a concern. For finding the first solution, the dominance of dynamic bundling over non-bundling is in general not guaranteed even in terms of nodes visited and constraint checks. In order to assess the time overhead for bundling and its impact on the search effort, we compare the median values of the following metrics for DynBndl and FC on radomly generated problems: the CPU time, the number of nodes visited `NV`, the number of constraint checks (`CC`), the compaction ratio defined as the ratio of number solution to the number of bundles `Comp-Ratio` (when finding all solutions), the number of additional solutions due to bundling FBS+ (when finding a first solution). Note that any savings by DynBndl in the number of nodes visited when looking for a first solution can only be explained by the bundling of no-goods.

Below, we report results demonstrating the benefits of dynamic bundling on randomly generated problems. We describe a non-binary CSP with the tuple $\langle k, n, a, p_2, c_3, c_4, t\rangle$, where $k$ is constraint arity, $n$ the number of variables, $a$ the domain size, $p_2$ the constraint probability of binary constraints, $c_3$ and $c_4$ are the number of ternary and quaternary constraint respectively, and $t$ the constraint tightness. We considered CSPs with $p_2$=0.25, $c_3$=3 and $c_4$=2. In our experience, real-world applications usually have many binary constraints and a few non-binary constraints, which justifies the constraint distribution we choose. We tested CSPs with domain sizes of $a$=$\{10, 15\}$ in order to study the effect of the domain size and to test the performance of bundling on larger search trees. We used the random generator provided by [20] without enforcing solvability. The results are of the experiments are shown as follows:

| **Figure** | **Experiment** | **Numbr. of samples** per data point |
|---|---|---|
| Figure 10 | Finding all solutions for both $a$=10 and $a$=15 | 150 |
| Figure 11 | Finding the first solution for $a$=10 | 350 |
| Figure 12 | Finding the first solution for $a$=15 | 350 |
| Figure 13 | Savings in CPU time and nodes visited for finding a first solution $a$=10 and $a$=15 | 350 |

[4] We test only forward checking.

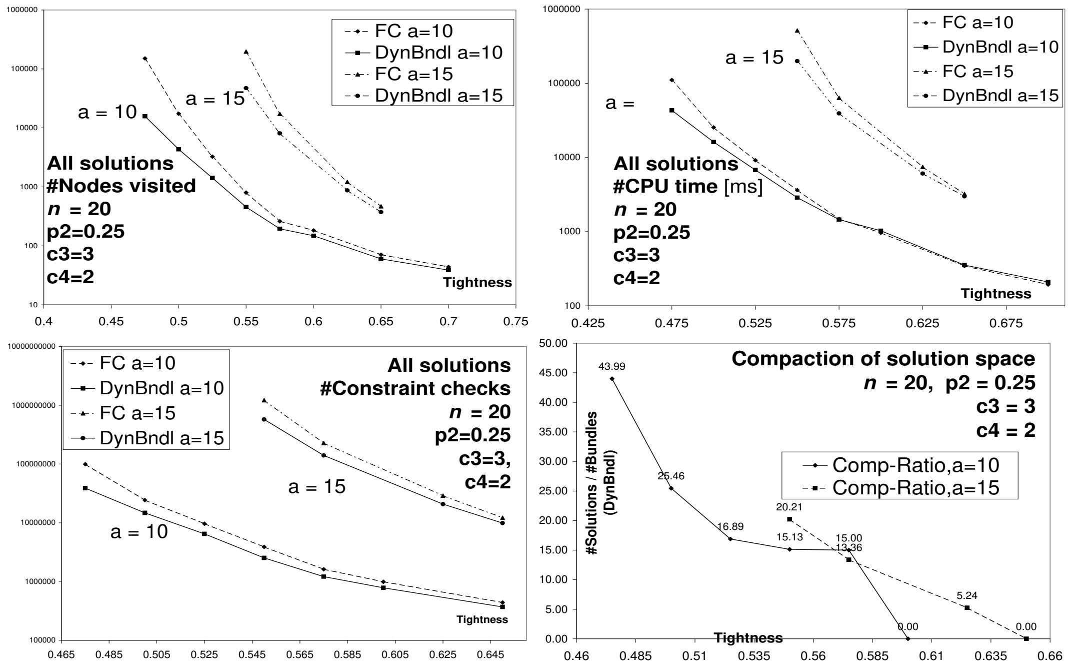


**Fig. 10.** *Comparison of DynBndl and FC for finding all solutions.*

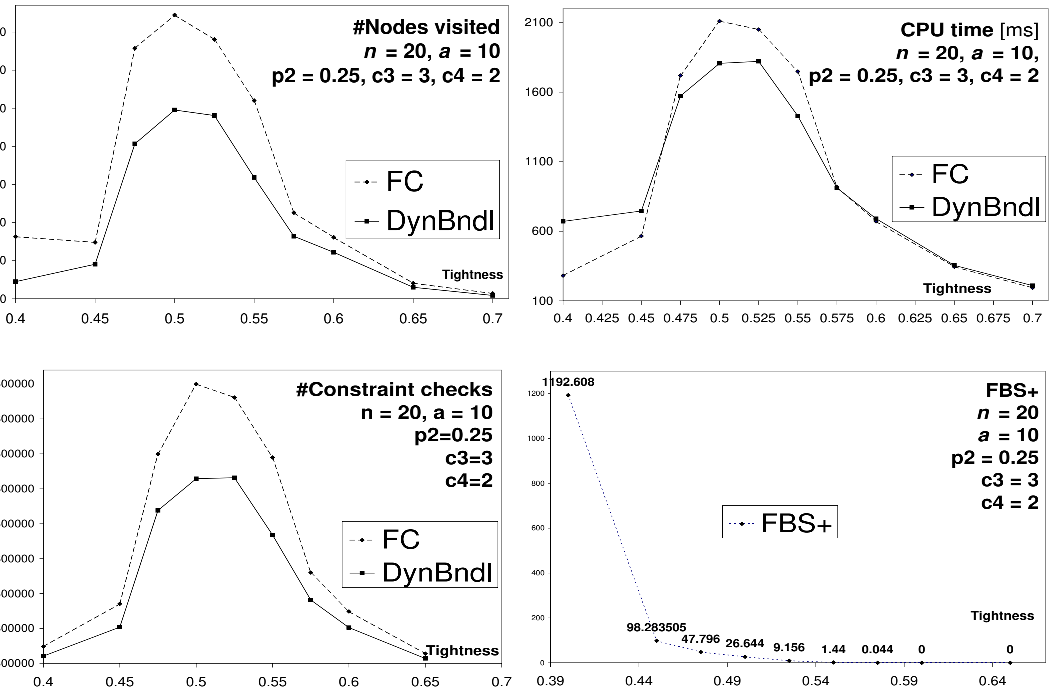


**Fig. 11.** *Comparing DynBndl and FC for finding one solution with a = 10.*

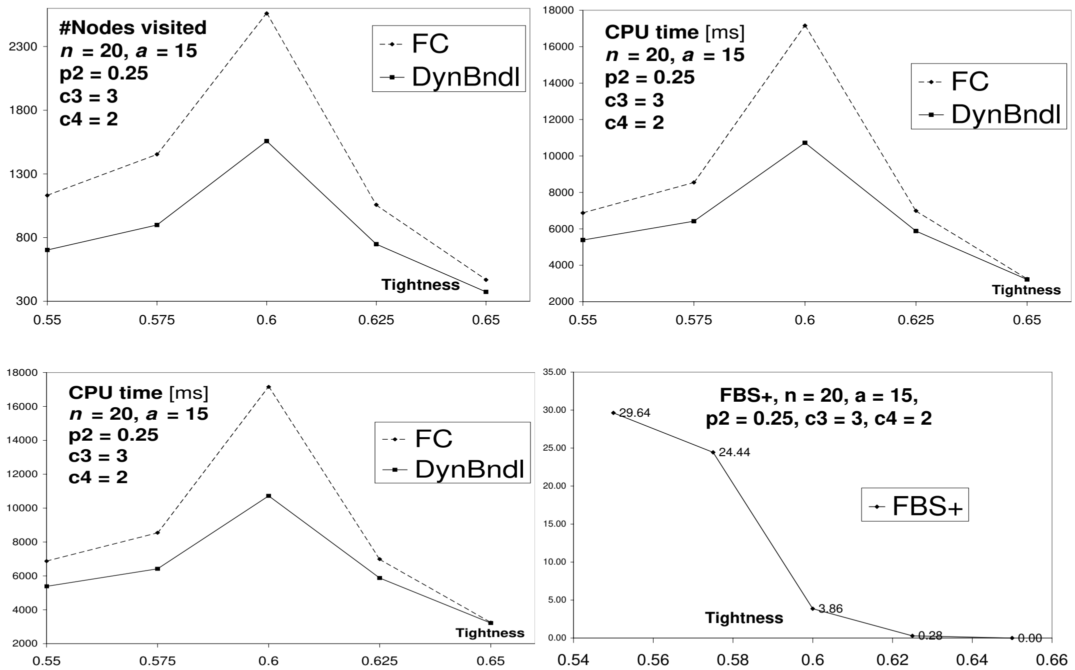


**Fig. 12.** *Comparing DynBndl and FC for finding one solution with a=15.*

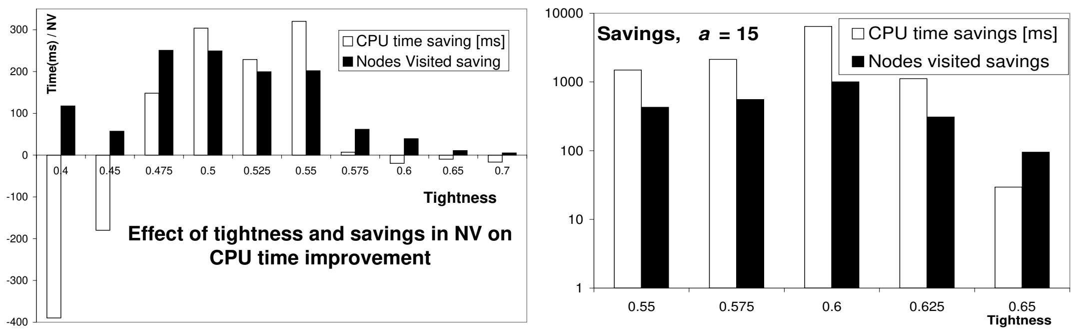


**Fig. 13.** *Savings in nodes visited and CPU time for a=10 (left) and a=15 (right).*

In the experiment of Figure 10, we had to restrict the values of tighteness tested because the experiments were prohibitively long. The results show that bundling outperforms non-bundling search when finding all solutions. Note the logarithmic scale on the $y$-axis for NV, CPU time, and CC. It is interesting to notice that DynBundl almost always reduces the CPU time, suggesting that effort for computing the bundles is negligible compared to the savings in the search effort. There is one exception for higher tightness values, where FC quickly detects the non-existence of solutions while bundling causes an overhead.

Figures 11 and 12 show results for finding one solution. In our analysis, we distinguish the performance at the following three tightness regions: low tightness, around the cross-over point, and high tightness.

- At low tightness values, FBS+ is large but the first solution is found without much backtracking. Therefore, bundling is useful for finding fat solutions but its benefits in bundling no-goods are not visible. The size of constraint definitions is the largest when tightness is low, thus increasing the overhead due to bundling. Although there are savings in NV and CC, the effort of finding domain partitions is not negligeable. This overhead is reflected by the CPU time measurements. The case of $t$=0.400 is an extreme where we get 1100 solutions at an additional cost of 270 milliseconds. This overhead is compensated by the large number number of alternative and robust solutions obtained by DynBndl.
- Around the cross-over point, bundling of no-goods by DynBndl becomes prevelant and we encounter the maximum amount of savings, including in CPU time. Here, the effort of computing bundles is insignificant compared to the savings due to bundling. With a larger domain size (i.e., comparing Figure 11 and Figure 12), the savings at the phase-transition region increase. Indeed, DynBndl records a higher percentage improvement for larger problems. At $a$=10 the improvement CPU time is 18% and in NV is 36%, while at $a$=15 the improvement in CPU time is 38% and in NV is 40%. This is explained as follows. The search tree with $a$=15 is larger than that for $a$=10, and therefore the number of nodes saved by DynBndl is much larger (though close in percentage) and the savings in CPU time due to visiting these many fewer nodes overshadows the overheads of bundling. Therefore, from these results we can conclude that in the phase-transition region DynBndl significantly improves the performnce of search, even for finding the first solution.
- For high tightness values, most of the CSPs are not solvable and forward checking effectively detects this unsolvability early on in the search process, thus reducing the number of nodes visited and the number of backtracks. As mentionned above, the overhead due to bundling becomes apparent, although not alamingly so.

Figure 13 stresses the observations made above, perhaps providing more insight into the working of DynBndl. Observe the savings for tightness values $t$=0.475 and $t$=0.500 for $a$=10. The saving in terms of nodes visites are comparable, however, the savings in terms of CPU time are higher for the higher tightness value. This can be explained by the fact that, for low tightness, the size of constraints is larger, which increases the cost bundling.

We ran pilot experiments using higher constraint probalities ($p_2$= 0.5) and found that the CSPs generated have little symmetries that DynBndl can exploit. The partitions found in the domain of a variable due to one constraint are likely to be broken by those found due to the other constraints that apply to the variable. In the presence of a large number of constraints, hardly any bundling is done early on in the search, thus decreasing the savings in nodes visited. This last test indicates that bundling is likely to be less effective for dense random CSPs than for sparse ones. However, because random problems are less likely to exhibit symmetries than real-world problems, the performance of dynamic bundling on real-world problems may still be competitive but remains to be investigated.

## 5 Conclusions and future work

In this paper, we described a technique for computing the neighborhood interchangeable values of a CSP variable in the presence of non-binary constraints, for which no techniques are reported in the literature to the best of our knowledge. We used this technique for dynamically bundling the search space of backtrack search and and gave empirical evidence of its effectiveness of dynamic bundling. The improvement is particularly significant in the region of the cross-over point, which is a most valuable. In the future, we plan on further improving our implementation and evaluate the effectivement of these techniques for solving real-world problems such as such custom document assembly [21] and query optimization using materialized views in databases.

*Acknowledgments.* 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.

## References

1. Freuder, E.C.: Eliminating Interchangeable Values in Constraint Satisfaction Problems. In: Proc. of AAAI-91, Anaheim, CA (1991) 227–233
2. Beckwith, A.M., Choueiry, B.Y., Zou, H.: How the Level of Interchangeability Embedded in a Finite Constraint Satisfaction Problem Affects the Performance of Search. In: AI 2001: Advances in Artificial Intelligence, $14th$ Australian Joint Conference on Artificial Intelligence. LNAI Vol. 2256, Adelaide, Australia, Springer Verlag (2001) 50–61
3. Choueiry, B.Y., Davis, A.M.: Dynamic Bundling: Less Effort for More Solutions. In Koenig, S., Holte, R., eds.: 5th International Symposium on Abstraction, Reformulation and Approximation (SARA 2002). Volume 2371 of Lecture Notes in Artificial Intelligence., Springer Verlag (2002) 64–82
4. Lal, A., Choueiry, B.Y.: Constraint Processing Techniques for Improving Join Computation: A Proof of Concept. In: Proceedings of the $1^{st}$ International Symposium on Applications of Constraint Databases, CDB'04. Volume 3074 of LNCS. (2004) 149–167
5. Glaisher, J.: On the Problem of the Eight Queens. Philosophical Magazine, series 4 **48** (1874) 457–467
6. Puget, J.F.: On the Satisfiability of Symmetrical Constrained Satisfaction Problems. In: ISMIS'93. (1993) 350–361
7. Ellman, T.: Abstraction via Approximate Symmetry. In: Proc. of the 13 $^{th}$ IJCAI, Chambéry, France (1993) 916–921

8. Haselböck, A.: Exploiting Interchangeabilities in Constraint Satisfaction Problems. In: Proc. of the 13 $^{th}$ IJCAI, Chambéry, France (1993) 282–287
9. Rossi, F., Petrie, C., Dhar, V.: On the Equivalence of Constraint Satisfaction Problems. In: Proc. of the 9 $^{th}$ ECAI, Stockholm, Sweden (1990) 550–556
10. Bacchus, F., van Beek, P.: On the Conversion between Non-Binary and Binary Constraint Satisfaction Problems Using the Hidden Variable Method. In: Proc. of AAAI-98, Madison, Wisconsin (1998) 311–318
11. Bessière, C., Meseguer, P., Freuder, E.C., Larrosa, J.: On Forward Checking for Non-binary Constraint Satisfaction. Artificial Intelligence **141 (1-2)** (2002) 205–224
12. Neagu, N., Faltings, B.: Exploiting Interchangeabilities for Case Adaptation. In: International Conference on Case-Based Reasoning (ICCBR 01), Vancouver, British Columbia, Canada (2001)
13. Petcu, A., Faltings, B.: Applying Interchangeability Techniques to the Distributed Breakout Algorithm. In AAAI, ed.: Proceedings of IJCAI 2003. (2003)
14. Benson, B.W., Freuder, E.C.: Interchangeability Preprocessing Can Improve Forward Checking Search. In: Proc. of the 10 $^{th}$ ECAI, Vienna, Austria (1992) 28–30
15. Hubbe, P.D., Freuder, E.C.: An Efficient Cross Product Representation of the Constraint Satisfaction Problem Search Space. In: Proc. of AAAI-92, San Jose, CA (1989) 421–427
16. Cheeseman, P., Kanefsky, B., Taylor, W.M.: Where the Really Hard Problems Are. In: Proc. of the 12 $^{th}$ IJCAI, Sidney, Australia (1991) 331–337
17. Sabin, D., Freuder, E.C.: Contradicting Conventional Wisdom in Constraint Satisfaction. In: Proc. of the 11 $^{th}$ ECAI, Amsterdam, The Netherlands (1994) 125–129
18. Lesaint, D.: Maximal Sets of Solutions for Constraint Satisfaction Problems. In: Proc. of the 11 $^{th}$ ECAI, Amsterdam, The Netherlands (1994) 110–114
19. Choueiry, B.Y., Noubir, G.: On the Computation of Local Interchangeability in Discrete Constraint Satisfaction Problems. In: Proc. of AAAI-98, Madison, Wisconsin (1998) 326–333 Revised version KSL-98-24, `ksl-web.stanford.edu/KSL_Abstracts/KSL-98-24.html`.
20. Lal, A., Choueiry, B.Y., Zou, H.: A Generator for Solvable Random Non-Binary Finite Constraint Satisfaction Problems. consystlab.unl.edu (2003)
21. Purvis, L.: A Genetic Algorithm Approach to Automated Custom Document Assembly. In: Proc. of the Intelligent Systems Design and Applications Conference. (2002) 131–136