# How the Level of Interchangeability Embedded in a Finite Constraint Satisfaction Problem Affects the Performance of Search

Amy M. Beckwith, Berthe Y. Choueiry, and Hui Zou

Department of Computer Science and Engineering, 115 Ferguson Hall,
University of Nebraska-Lincoln, Lincoln NE 68588-0115
{abeckwit | choueiry | hzou}@cse.unl.edu

**Abstract.** We investigate how the performance of search for solving finite constraint satisfaction problems (CSPs) is affected by the level of interchangeability embedded in the problem. First, we describe a generator of random CSPs that allows us to control the level of interchangeability in an instance. Then we study how the varying level of interchangeability affects the performance of search for finding one solution and all solutions to the CSP. We conduct experiments using forward-checking search, extended with static and dynamic ordering heuristics in combination with non-bundling, static, and dynamic bundling strategies. We demonstrate that: (1) While the performance of bundling decreases in general with decreasing interchangeability, this effect is muted when finding a first solution. (2) Dynamic ordering strategies are significantly more resistant to this degradation than static ordering. (3) Dynamic bundling strategies perform overall significantly better than static bundling strategies. Even when finding one solution, the size of the bundles yielded by dynamic bundling is large and less sensitive to the level of interchangeability. (4) The combination of dynamic ordering heuristics with dynamic bundling is advantageous. We conclude that this combination, in addition to yielding the best results, is the least sensitive to the level of interchangeability, and thus, indeed is superior to other searches.

## 1 Introduction

A Constraint Satisfaction Problem (CSP) [12] is the problem of assigning values to a set of variables while satisfying a set of constraints that restrict the allowed combinations of values for variables. In its general form, a CSP is NP-complete, and backtrack search remains the ultimate technique for solving it. Because of the flexibility and expressiveness of the model, Constraint Satisfaction has emerged as a central paradigm for modeling and solving various real-world decision problems in computer science, engineering, and management.

It is widely acknowledged that real-world problems exhibit an intrinsic non-random structure that makes most instances 'easy' to solve. When the structure of a particular problem is known in advance, it can readily be embedded in the model and exploited during search [3], as it is commonly done for the pigeon-hole

problem. A challenging task is to *discover* the structure in a particular problem instance. In our most recent research [5, 2, 4], we have investigated mechanisms for discovering and exploiting one particular type of symmetry structure, called *interchangeability*, that allows us to bundle solutions, and have integrated these *bundling* mechanisms with backtrack search. We have investigated and evaluated the effectiveness of this integration and demonstrated its utility under particularly adverse conditions (i.e., random problems generated without any particular structure embedded *a priori* and puzzles known to be extraordinarily resistant to our symmetry detection techniques). In this paper, we investigate how the performance of these new search strategies is affected by the level of interchangeability embedded in the problem. We first show how to generate random problems with a controlled level of inherent structure, then demonstrate the effects of this structure on the performance of the various search mechanisms with and without interchangeability detection.

Section 2 gives a brief background to the subject and summarizes our previous work. Section 3 describes our random generator, designed to create random CSP instances with a pre-determined level of interchangeability. Section 4 introduces the problem sets used for testing, and demonstrates the performance of our search strategies across varying levels of interchangeability. Section 5 concludes the paper with directions for future research.

## 2 Background and contributions

A finite Constraint Satisfaction Problem (CSP) is defined as $\mathcal{P}=(\mathcal{V}, \mathcal{D}, \mathcal{C})$; where $\mathcal{V}=\{V_1, V_2, \ldots, V_n\}$ is a set of variables, $\mathcal{D}=\{D_{V_1}, D_{V_2}, \ldots, D_{V_n}\}$ is the set of their corresponding domains (the domain of a variable is a set of possible values), and $\mathcal{C}$ is a set of constraints that specifies the acceptable combinations of values for variables. A solution to the CSP is the assignment of a value to each variable such that all constraints are satisfied. The question is to find one or all solutions. A CSP is often represented as a constraint (hyper-)graph in which the variables are represented by nodes, the domains by node labels, and the constraints between variables by (hyper-)edges linking the nodes in the scope of the corresponding constraint. We study CSPs with finite domains and binary constraints (i.e., constraints apply to two variables).

Since a general CSP is NP-complete, it is usually solved by backtrack search, which is an exponential procedure. We enhance this basic backtrack search through the identification and exploitation of structure in the problem instance. This structure is in the form of symmetries. In particular, we make use of a type of symmetry called interchangeability, which was introduced and categorized by Freuder in [7]. We limit our investigations to interchangeability among the values in the domain of one given variable. Interchangeability between two values for the variable exists if the values can be substituted for one another without affecting the assignments of the remaining variables. Two such values are said to belong to the same *equivalence class*. Each equivalence class is a bundle of values that can be replaced by one representative of the bundle, thus reducing

the size of the initial problem. We call the number of distinct equivalence classes in the domain of a variable the degree of *induced domain fragmentation*, `IDF`.

Freuder [7] proposed an efficient algorithm, based on building a discrimination tree, for computing one type of interchangeability, neighborhood interchangeability (NI). NI partitions the domain of a variable into equivalence classes given all the constraints incident to that variable. Haselböck [11] simplified NI to a weaker form that we call *neighborhood interchangeability according to one constraint* ($\mathrm{NI}_C$). He showed how to exploit $\mathrm{NI}_C$ advantageously in backtrack search (BT), with and without forward-checking (FC) for finding all the solutions of a CSP. He also showed how $\mathrm{NI}_C$ groups multiple solutions of a CSP into *solution bundles*. In a solution bundle, each variable is assigned a specific subset of its domain instead of the unique value usually assigned by backtrack search. Any combination of one value per variable in the solution bundle is a solution to the CSP. Such a bundle not only yields a compact representation of this solution set, but is also useful in the event that one component of a solution fails, and an alternate, equivalent solution must be found quickly. In the bundling strategy proposed by Haselböck, symmetry relations are discovered *before* search is started. These are *static* interchangeability relations. We refer to this strategy as *static bundling*. Below we summarize our previous results [5, 2, 4], which motivate the investigations we report here.

In [5], we proposed to compute interchangeability *dynamically during* search using a generalized form of Freuder's discrimination tree, the joint discrimination tree of Choueiry and Noubir [6]. We called this type of interchangeability *dynamic neighborhood partial interchangeability* (DNPI). Since DNPI is computed during search, we say that it performs *dynamic bundling*. DNPI induces less domain fragmentation (larger partitions) than $\mathrm{NI}_C$ and is thus likely to find larger solution bundles. We designed a new search strategy that combines dynamic bundling (DNPI) with forward-checking, and compared it to searches without bundling and with static bundling ($\mathrm{NI}_C$) for forward-checking search, see Fig. 1. We proved that the relations shown in Fig. 2 (left) hold when searching for all so-

| **Search** | | **Comparison criteria** |
|---|---|---|
| Non Bundling [10] | FC | Number of constraint checks `CC`, nodes visited `NV`, solution bundles `SB`, and CPU time. |
| Static bundling [11] | $\mathrm{NI}_C$ | |
| Dynamic bundling [5] | DNPI | |

**Fig. 1.** *Search and bundling strategies.*

lutions (provided the variable and value orderings are the same for all searches), thus establishing that dynamic bundling is *always* worthwhile when solving for all solutions. In addition to the theoretical guarantees of Fig. 2 (left), we showed empirically that neither non-bundling (FC) nor static bundling ($\mathrm{NI}_C$) search outperforms dynamic bundling search in terms of the quality of bundling (i.e., number of solution bundles generated) and in terms of the standard comparison criteria for search (i.e., number of constraint checks and number of nodes visited). CPU time measurements were reasonably in-line with the other criteria.

In [2], we modified the forward-checking backtrack-search procedures of Fig. 1 to allow the integration of *dynamic* variable-value orderings with bundling strategies, while looking for all solutions. We examined the following ordering heuris-

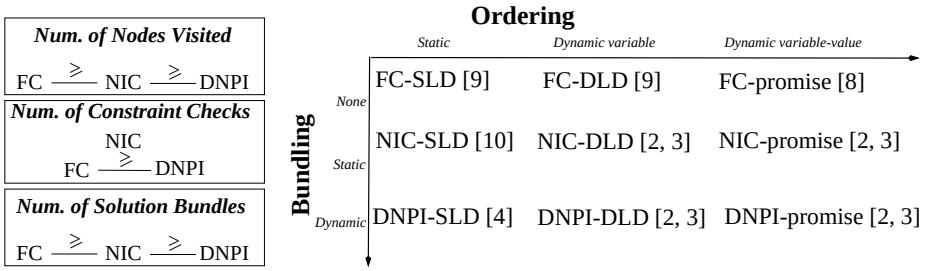


**Fig. 2.** *Left:* Comparison of search strategies assuming the same variable orderings for all strategies and while looking for all solutions. *Right:* Interleaving dynamic bundling with dynamic ordering.

tics: (1) static least-domain (`SLD`), (2) dynamic least-domain (`DLD`) and (3) dynamic variable-value ordering (`promise` of Geelen [9]). The search algorithms generated fell into the nine categories shown in Fig. 2 (right). Since the variable and value orderings can no longer be maintained across strategies, strong, theoretical results similar to the ones of Fig. 2 (left) cannot be made. We instead make empirical evaluations. Our experiments on these nine search strategies showed that dynamic least-domain ordering combined with dynamic bundling (DNPI-DLD) almost always yields the most effective search and the most compact solution space. Further, we noted that although `promise` reduces significantly the number of nodes visited in the search tree, it is harmful in the context of searching for all solutions because the number of constraint checks it requires is prohibitively large[1].

Finally, in [4], we addressed the task of finding a *first* solution. In addition to the ordering heuristics listed above (i.e., `SLD`, `DLD`, and `promise`), we proposed and tested two other ordering heuristics, specific to bundling: (1) Least-Domain-Max-Bundle (`LD-MB`) chooses the variable of smallest domain and, for this variable, the largest bundle in its domain; and (2) Max-Bundle (`Max-Bundle`) chooses the largest available bundle among all bundles of all variables. We found that the `promise` heuristic of Geelen [9] performs particularly well for finding one solution, consistently finding the largest first bundle with loosest bottlenecks[2], and nearly always yielding a backtrack-free search. This must be contrasted to its bad performance for finding all solutions in [2]. Further, dynamic bundling again proved to outperform static bundling, especially when used in combination with `promise`. Finally, we noted that `LD-MD`, our proposed new heuristic, is competitive with relatively few constraint checks, low CPU time, and good bundling.

The above summarized research established the utility of discovering and exploiting interchangeability relationships in general. In all of our past work, algorithms were tested on CSPs created with the random generator of Bacchus and van Run [1], which did not intentionally embed any structure in the problems. This paper furthers our investigation of interchangeability and adds the following contributions:

[1] The `promise` heuristic is by design best suited for finding one solution [9].
[2] The bottleneck of a solution bundle is the size of the smallest domain in the bundle.

1. We introduce a generator of random CSPs that allows us to control the level of interchangeability embedded in a problem in addition to controlling the size of the CSP, and the density and tightness of the constraints.
2. Using this generator, we conduct experiments that test the previously listed search strategies[3] across various levels of interchangeability. See Table 1:

| Problem | Bundling | Ordering |
|---|---|---|
| Finding all solutions | × {FC, $NI_C$, DNPI} | × {SLD, DLD} |
| Finding first solution | × {FC, $NI_C$, DNPI} | × {SLD, DLD, LD-MB [4], promise [9]} |

**Table 1.** *Search strategies tested.*

3. We show that: (a) Both static and dynamic bundling search strategies do indeed detect and benefit from interchangeability embedded in a problem instance. (b) the performance of dynamic bundling is significantly superior to that of static bundling when looking for a first solution bundle. (c) Problems with embedded interchangeability are not easier, or more difficult, to solve for the naive FC algorithm. And (d) Most algorithms are affected by the variance of interchangeability. However, DLD-ordered search is less sensitive and performs surprisingly well in all situations.

## 3 A generator that controls interchangeability

Typically, a generator of random binary CSPs takes as input the following parameters $\langle n, a, p, t\rangle$. The first two parameters, $n$ and $a$ relate to the variables—$n$ gives the number of variables, and $a$ the domain size of each variable. The second two parameters, $p$ and $t$ control the constraints—$p$ gives the probability that a constraint exists between any two variables (which also determines the number of constraints in the problem $C = p\frac{n(n-1)}{2}$), and $t$ gives the constraint tightness (defined as the ratio of the number of tuples disallowed by the constraint over all possible tuples between the two variables).

In order to investigate the effects of interchangeability on the performance of search for solving CSPs, we must guarantee from the outset that each CSP instance contains a specific, controlled amount of interchangeability. Interchangeability within the problem instance is determined by the constraints. Indeed each constraint fragments the domain of the variable to which it applies into equivalence classes (as discussed below, Fig. 3) that can be exploited for bundling. Therefore, the main difficulty in generating a CSP for testing bundling algorithms resides in the generation of the constraints. We introduce an additional parameter to our random generator [14] that controls the number of equivalence

[3] LD-MB for finding all solutions collapses to DLD. Because of their poor behavior [2, 4], we exclude from our current experiments: (1) Max-Bundle for finding a first solution and (2) all dynamic strategies for variable-value orderings (e.g., promise and Max-Bundle) for finding all solutions.

classes induced by a constraint. This parameter, IDF, provides a measure of the interchangeability in a problem: a higher IDF means less interchangeability. In compliance with common practices, our generator adopts the following standard design decisions: (1) All variables have the same domain size and, without loss of generality, the same values. (2) Any particular pair of variables has only one constraint. (3) All constraints have the same degree of induced domain fragmentation. (4) All constraints have the same tightness. And, (5) any two variables are equally likely to be connected by a constraint.

### 3.1 Constraint representation and implementation

A constraint that applies to two variables is represented by a *binary matrix* whose rows and columns denote the domains of the variables to which it applies. The '1' entries in the matrix specify the tuples that are allowed and the '0' entries the tuples that are disallowed. Fig. 3 shows a constraint $c$, with $a = 5$ and $t = 0.32$. This constraint applies to $V_1$ and $V_2$ with domains $\{1, 2, 3, 4, 5\}$. The matrix is implemented as a list of row-vectors. Each row corresponds to a value in the domain of $V_1$. Each constraint partitions the domains of the variables

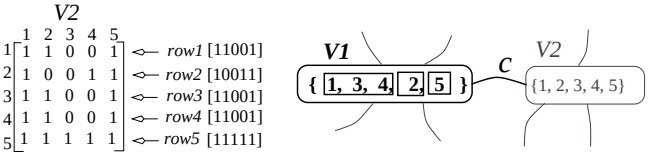


**Fig. 3.** *Constraint representation as a binary matrix.* Left: Encoding as row vectors. Right: Domain of $V_1$ partitioned by interchangeability.

to which it applies into equivalence classes. The values in a given equivalence class of a variables are consistent with the same set of values in the domain of the other variable. Indeed, $c$ fragments the domain of $V_1$ into three equivalence classes corresponding to rows $\{1, 3, 4\}$, $\{2\}$ and $\{5\}$ as shown in Fig. 3.

We define the *degree of induced domain fragmentation* (IDF) of a constraint as the number of equivalence classes it induces on the domain of the variable *whose values index the rows of the matrix*. Thus the degree of induced domain fragmentation of $c$ for $V_1$ is $\texttt{IDF} = 3$. Since we control the IDF for only one of the variables (the one represented in the rows), our constraints are not *a priori* symmetrical. The domain fragmentation induced on the remaining variable is not controlled. Our generator constitutes an improvement of the random generator with interchangeability of Freuder and Sabin [8], which inspired us. The latter creates each constraint from the conjunction of two components: one component controlling interchangeability and the other controlling tightness. The component controlling interchangeability is both symmetrical and non-reflexive (i.e., all diagonal entries in the matrix are 0). Therefore, both variables in a binary constraint have the same degree of induced domain fragmentation. This symmetry may affect the generality of the resulting constraint. Indeed, Freuder and Sabin introduce the second component of their constraint in order to achieve more generality and to control constraint tightness. This second component is a

random constraint with a specified tightness $t$. The resulting constraint obtained by making the conjunction of the two components is likely to be tighter than specified and also contain less interchangeability than specified. To avoid this problem, our generator first generates a constraint with a specified tightness, imposes the degree of IDF requested, then checks the resulting tightness. Thus, in the CSPs generated, we guarantee that *both* $t$ and IDF meet the specifications without sacrificing generality.

### 3.2 Constraint generation

Constraint generation is done according to the following five-step process:

**Step 1:** *Matrix initialization.* Create an $a \times a$ matrix with every entry set to 1.
**Step 2:** *Tightness.* Set random elements of the matrix to 0 until specified tightness is achieved.
**Step 3:** *Interchangeability.* Modify the matrix to comply with the specified degree of induced domain fragmentation, see below.
**Step 4.** *Tightness check.* Test the matrix. If tightness meets the specification, continue. Otherwise, throw this matrix away and go to Step 1.
**Step 5.** *Permutation:* Randomly permute the rows of the generated matrix.

When $C$ constraints have been successfully generated ($C = p\frac{n(n-1)}{2}$), each constraint is assigned to a distinct random pair of variables. Note that we do not impose any structure on the generated CSP other than controlling the IDF in the definition of the constraints. We also do not guarantee that the CSP returned is connected. However, when $C > n - 1$ extensive and random checks detected no unconnected CSPs among the ones generated. Obviously, when $C > \frac{(n-1)(n-2)}{2}$, connectedness is guaranteed. Below, we describe in further detail Steps 3 and 5 of the above process. Steps 1, 2, and 4 are straightforward.

**Step 3: Achieving the degree of induced domain fragmentation (IDF).** After generating a matrix with a specific tightness, we compute its IDF by counting the number of distinct row vectors. Each vector is assigned to belong to a particular induced equivalence class. In the matrix of Fig. 3, $row1$, $row3$ and $row4$ would be assigned to the equivalence class 1, $row2$ assigned to equivalence class 2, and $row5$ assigned to equivalence class 3. When the value of IDF requested different from that of the current matrix, we modify the matrix to increase or decrease its IDF by one as discussed below until meeting the specification. To increase IDF, we select any row from any equivalence class that has more than one element and make it the only element of a new equivalence class. This is done by randomly swapping distinct bits in the vector selected until obtaining a vector distinct from all other rows. Note this operation does not modify the tightness of the constraint. To decrease IDF, we select a row that is the only element of an equivalence class and set it equal to any another row. For example in Fig 3, setting $row2 \leftarrow row5$ decreases IDF from 3 to 2. This operation may affect tightness. When this is complete, Step 4 verifies that the

tightness of the constraint has not changed. If it has, we start over again, generating a new constraint. If the tightness is correct, we proceed to the following step, *row permutation*.

**Step 5: Row permutation.** In order to increase our chances of generating random constraints and avoid that the fragmentation determined by one constraint on the domain of a variable coincidences with that induced by another constraint, the rows of each successfully generated constraint are permuted. The permutation process chooses and swaps random rows a random number of times. The input and output matrices of this process obviously have the same tightness and interchangeability as this process does not change these characteristics.

### 3.3 Constraint generation in action

An example of this 5-step process is shown in Figure 4, where we generate a constraint for $a = 5$, $\texttt{IDF} = 3$ and $t = 0.32$. Note that Step 3 and Step 4, which

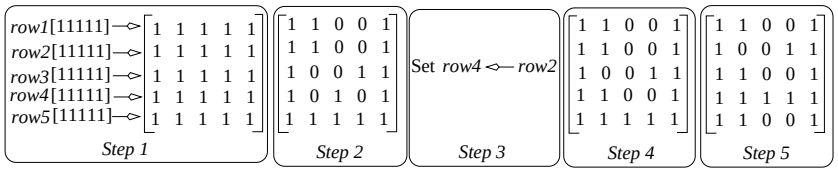


**Fig. 4.** *Constraint generation process.*

control the interchangeability and tightness of a matrix may fail to terminate successfully. This happens when: (1) No solution exists for the combination of the input parameters. It is easy to check that when $a = 5, t = 0.04$, there exists only solutions with $\texttt{IDF} = 2$, due to the presence of only one 0 in the matrix. And (2) although a solution may exist, the process of modifying interchangeability in the matrix continuously changes tightness. To avoid entering an infinite loop in either of these situations, we use a counter at the beginning of the process of constraint generation. After 50 attempts to generate a constraint, it times out, and the generation of the current CSP is interrupted. Our current implementation of the generator exhibits a failure rate below 5%, and guarantees constraints with both the specified tightness and degree of induced domain fragmentation.

## 4 Tests and Results

We generated two pools of test problems using our random generator, each with a full range of values for $\texttt{IDF}$, $t$, and $p$ and 20 instances per measurement point. The first pool has the following input parameters: $n = 10$, $a = 5$, $p = [.1, 1.0]$ with a step of 0.1, $\texttt{IDF} = 2, 3, 4, 5$, and $t = [.04, .92]$, with a step of 0.08. The second pool has the input parameters: $n = 10$, $a = 7$, $p = [.1, 0.9]$ with a step of 0.2, $\texttt{IDF} = 2, 3, \ldots, 7$, and $t = [0.04, 0.92]$, with a step of 0.16. Recall that when $p$ is small, the CSP is not likely to be connected, and when $p = 1$, the CSP is a complete graph. Note that instances with $\texttt{IDF} = a$ have *no* embedded interchangeability

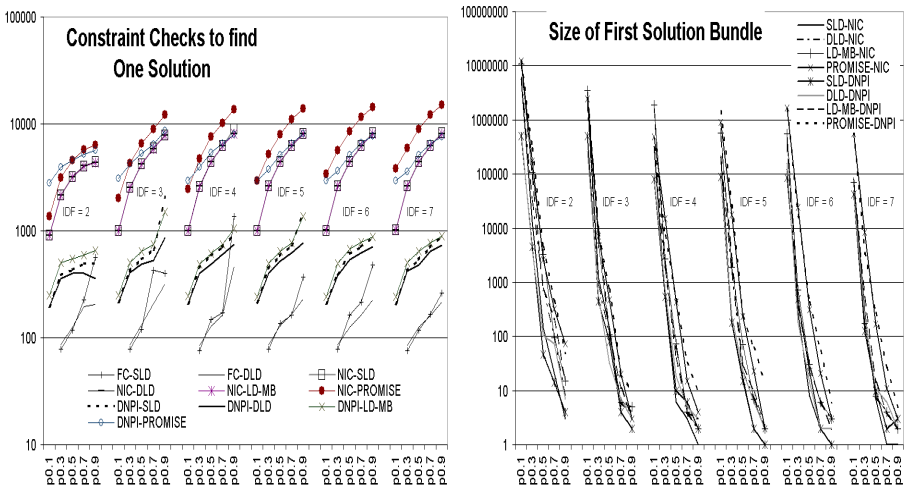


**Fig. 5.** *Comparing performance of search for finding one solution,* $t = 0.28$.

and thus provide the *most adverse conditions* for bundling algorithms. We tested the strategies of Table 1 on each of these two pools, and took the averages and the median values of the number of nodes visited (`NV`), constraint checks (`CC`), size of the bundled solution (when finding the first bundle), number of solution bundles (when finding all solutions), and CPU time. We report here only the average values since the median are qualitatively equivalent.

Constraint tightness has a large effect on the solvability of a random problem. Problems with loose constraints are likely to have many solutions. As tightness grows, the values of all measured parameters (`CC`, `NV`, CPU time, and bundle size) quickly die to zero because almost all problems become unsolvable (especially for $t \geq 0.5$). The behavior of the various algorithms is best visible at relatively low values for tightness. In Fig. 5 and Fig. 6 we display charts for tightness values of $t = 0.28$ with the second problem pool, where each variable has a domain size of 7 ($a = 7$). The patterns observed on this data set shown are consistent across all values for tightnesses for both problem pools and are not reported here for lack of space. Both figures show that the algorithms are affected by the increasing `IDF`. This effect is more visible in Fig. 6. This demonstrates that our generator indeed allows us to control of the level of interchangeability.

### 4.1 Finding the first bundle

In our experiments for finding the first solution bundle, we report in Fig. 5 the charts for `CC` (left) and bundle size (right). Note the logarithmic scale of both charts. The the chart for CPU time is similar to that of `CC` and is not shown.

Three of the four DNPI-based searches (DNPI-`promise` is the exception) reside toward the bottom of Fig. 5 proving DNPI performs better than $\text{NI}_C$ in terms of the search effort measured as `CC` (left), CPU time (not shown), and size of the first bundle (right). DNPI seems also more resistant than $\text{NI}_C$ to an

increasing IDF. Even in the absence of embedded interchangeability (large IDF) and when density is high (large $p$), DNPI-based strategies still perform some bundling performed (bundle size $> 1$).

At the left Fig. 5, FC is shown slightly below DNPI at the bottom of the chart. One is tempted to think that FC outperforms all the bundling algorithms. However, recall that FC finds only one solution, while DNPI finds from 5 up to one million solutions per bundle for a cost of at most 5 times that of FC. Furthermore, DNPI is finding not only a multitude of solutions, but the similarity of these solutions makes particularly desirable in practical applications for updating solutions.

### 4.2 Finding all solutions

The effects of increasing IDF are more striking when finding all solutions and are shown in Fig. 6. It is easy to see in all four charts of Fig. 6 that both static ($\text{NI}_C$) and dynamic (DNPI) bundling searches naturally perform better where there is interchangeability (low values of IDF) than when there is not (IDF approaches $a$). However, this behavior is much less drastic for DLD-based searches, which are less sensitive to the increase of IDF than SLD-based searches. Indeed the curves for DLD (both $\text{NI}_C$ and DNPI) rise significantly slower than its SLD counterparts as the value of IDF increases. Additionally, we see here more clearly than reported in [2], that search with DLD outperforms search with SLD for all evaluation criteria and for all values of $p$ and IDF.

From this data, one is tempted to think that the problems with high interchangeability (e.g., IDF $= 2$) are easier to solve in general than those with higher values of IDF. This is by no means the case. Our experiments have shown that non-bundling FC is not only insensitive to interchangeability, but also performs consistently several orders of magnitude worse than DNPI and $\text{NI}_C$. This data is not shown because it is 3 to 7 orders of magnitude larger than the other values.

Even when interchangeability was specifically not included in a problem (IDF $= a$), all bundling strategies, more significantly dynamic bundling, were able to bundle the solution space. This is due to the fact that as search progresses, some values are eliminated from domains, and thus more interchangeability may become present. This establishes again the superiority of dynamic bundling even in the absence of explicit interchangeability: its runtime is far faster than FC, and its bundling capabilities are clear.

## 5 Conclusions and directions for future research

In this paper we describe a generator of random binary CSPs that allows us to embed and control the structure, in terms of interchangeability, of a CSP instance. We then investigate the effects of the level of interchangeability on the performance of forward-checking search strategies that are perform no bundling (FC) and that exploit static ($\text{NI}_C$) and dynamic (DNPI) bundling. These strategies are combined with the most common or best performing ordering heuristics.

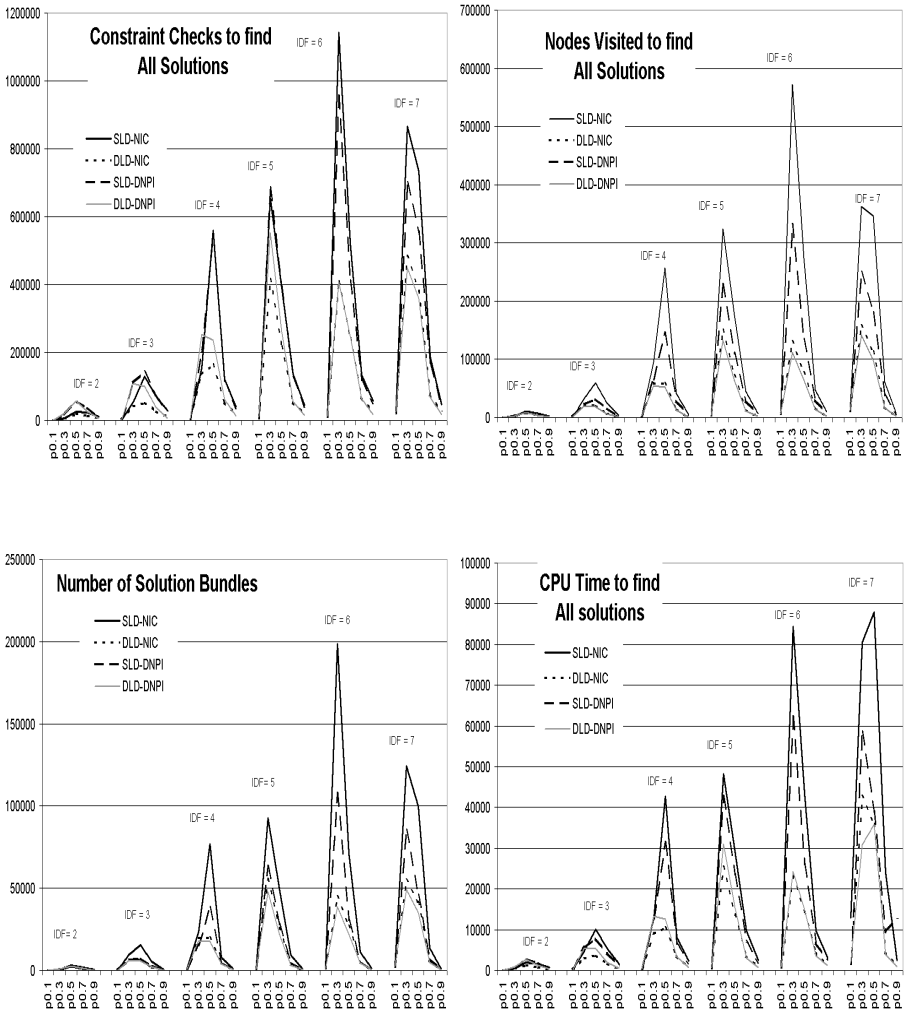


**Fig. 6.** *Comparing performance of search for finding all solutions,* $t = 0.28$.

We demonstrate that dynamic bundling strategies remain effective across all levels of interchangeability, even under particularly adverse conditions (i.e., `IDF` = domain size). While search with either static or dynamic ordering is able to detect and exploit the structure embedded in a problem, `DLD`-ordered search is less sensitive to the absence of interchangeability, performing quite well in all situations. In particular, we see that DNPI-DLD reacts slowly to the presence or absence of interchangeability while performing consistently well for finding either one or all solutions.

We intend to extend these investigations to non-binary CSPs and also to demonstrate that dynamic bundling may benefit from maintaining arc-consistency (MAC) of Sabin and Freuder [13]. Additionally, the flatness of the curves for

DNPI in Fig. 5 (left) makes us wonder how search strategies based on bundling may be affected by the famous phase-transition phenomenon.

## References


1. Fahiem Bacchus and P. van Run. Dynamic Variable Ordering in CSPs. In *Principles and Practice of Constraint Programming, CP'95. Lecture Notes in Artificial Intelligence 976*, pages 258–275. Springer Verlag, 1995.
2. Amy M. Beckwith and Berthe Y. Choueiry. Effects of Dynamic Ordering and Bundling on the Solution Space of Finite Constraint Satisfaction Problems. Technical Report CSL-01-03. `http://consystlab.unl.edu/CSL-01-03.ps`, University of Nebraska-Lincoln, 2001.
3. Cynthia A. Brown, Larry Finkelstein, and Paul W. Purdom, Jr. Backtrack Searching in the Presence of Symmetry. In T. Mora, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 99–110. Springer-Verlag, 1988.
4. Berthe Y. Choueiry and Amy M. Beckwith. On Finding the First Solution Bundle in Finite Constraint Satisfaction Problems. Technical Report CSL-01-03. `http://consystlab.unl.edu/CSL-01-04.ps`, University of Nebraska-Lincoln, 2001.
5. Berthe Y. Choueiry and Amy M. Beckwith. Techniques for Bundling the Solution Space of Finite Constraint Satisfaction Problems. Technical Report CSL-01-02. `http://consystlab.unl.edu/CSL-01-02.ps`, University of Nebraska-Lincoln, 2001.
6. Berthe Y. Choueiry and Guevara Noubir. On the Computation of Local Interchangeability in Discrete Constraint Satisfaction Problems. In *Proc. of AAAI-98*, pages 326–333, Madison, Wisconsin, 1998. Revised version KSL-98-24, `http://ksl-web.stanford.edu/KSL_Abstracts/KSL-98-24.html`.
7. Eugene C. Freuder. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *Proc. of AAAI-91*, pages 227–233, Anaheim, CA, 1991.
8. Eugene C. Freuder and Daniel Sabin. Interchangeability Supports Abstraction and Reformulation for Multi-Dimensional Constraint Satisfaction. In *Proc. of AAAI-97*, pages 191–196, Providence, Rhode Island, 1997.
9. Pieter Andreas Geelen. Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems. In *Proc. of the 10 $^{th}$ ECAI*, pages 31–35, Vienna, Austria, 1992.
10. Robert M. Haralick and Gordon L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.
11. Alois Haselböck. Exploiting Interchangeabilities in Constraint Satisfaction Problems. In *Proc. of the 13 $^{th}$ IJCAI*, pages 282–287, Chambéry, France, 1993.
12. Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.
13. Daniel Sabin and Eugene C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proc. of the 11 $^{th}$ ECAI*, pages 125–129, Amsterdam, The Netherlands, 1994.
14. Hui Zou, Amy M. Beckwith, and Berthe Y. Choueiry. A Generator of Random Instances of Binary Finite Constraint Satisfaction Problems with Controllable Levels of Interchangeability. Technical Report CSL-01-01. `http://consystlab.unl.edu/CSL-01-01.doc`, University of Nebraska-Lincoln, 2001.