Combining Ordering Heuristics and Bundling Techniques for Solving Finite Constraint Satisfaction Problems

Amy Beckwith
University of Nebraska-Lincoln, abeckwit@cse.unl.edu

Berthe Y. Choueiry
University of Nebraska-Lincoln, choueiry@cse.unl.edu

Follow this and additional works at: http://digitalcommons.unl.edu/csetechreports

Part of the Computer Sciences Commons

http://digitalcommons.unl.edu/csetechreports/121

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in CSE Technical reports by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.
Abstract

We investigate techniques to enhance the performance of backtrack search procedure with forward-checking (FC-BT) for finding all solutions to a finite Constraint Satisfaction Problem (CSP). We consider ordering heuristics for variables and/or values and bundling techniques based on the computation of interchangeability. While the former methods allow us to traverse the search space more effectively, the latter allow us to reduce its size. We design and compare strategies that combine static and dynamic versions of these two approaches. We show empirically the utility of dynamic variable ordering combined with dynamic bundling in both random problems and puzzles.
Contents

1 Introduction 3
2 Interchangeability 4
3 Search strategies 5
   3.1 Forward checking ........................................ 5
   3.2 Dynamic variable ordering ............................. 6
   3.3 Dynamic variable-value ordering ..................... 7
   3.4 Static bundling with FC-NIC ........................... 8
   3.5 Dynamic bundling with FC-DNPI .................... 8
4 New hybrid algorithms 9
5 Experiments 14
   5.1 Results and analysis .................................... 14
6 Conclusion and future work 15

List of Tables

1 Results on random problems. .............................. 16
2 Plots for random problems and table for puzzles. .... 17

List of Figures

1 Basic search algorithms. .................................. 5
2 Example of a dld search tree. ............................ 6
3 A possible search tree with dynamic variable-value ordering. ... 7
4 Search Algorithms. ........................................ 9
5 Finding the next variable to expand using dld. .......... 11
6 Finding the next variable to expand using promise in FC-NIC. ... 12
7 Finding the next variable to expand using promise in FC-DNPI. ... 13
1 Introduction

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 \( \mathcal{C} \) 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., they apply to two or fewer variables).

Since a general CSP is NP-complete, it is usually solved by search, which is an exponential procedure. Several strategies can be used to improve the performance of the search process. In this paper, we discuss the combination of two such improvements for finding and representing all solutions to a CSP. The first means to improve performance is based on ordering the variables and/or values dynamically during search, which improves the rate at which solutions are found. The second is the exploitation of interchangeabilities, which reduces the size of the search space by eliminating redundancies and yields a space of bundled solutions.

In this paper, we conduct experimental evaluations of three different ordering heuristics: namely, static variable ordering (with static least-domain, \( sld \)), dynamic variable ordering (with dynamic least-domain, \( dld \)), and dynamic variable-value ordering (with \( \text{promise} \) [Geelen 1992]). We combine each of these heuristic with standard backtrack search with forward checking and two bundling strategies, one static [Haselböck 1993] and one dynamic [Choueiry and Beckwith 2001]. We evaluate each of these combinations on a battery of puzzles and randomly generated problems.

We report the following contributions: (1) We provide an adaptation of the backtrack-search procedure to allow dynamic variable-value orderings with interchangeability, and (2) We demonstrate empirically that dynamic least-domain combined with dynamic bundling almost always yields the most effective search and the most compact solution space and (3) that although \( \text{promise} \) reduces significantly the number of nodes visited in tree, it is harmful in this context because of the significant increase of the number of constraint checks.

This paper is organized as follows. Section 2 summarizes the main concepts of interchangeability. Section 3 recalls the mechanisms of ordering heuristics and exploiting interchangeability in search. Section 4 gives techniques and pseudocode for generating the hybrid strategies. Section 5 describes our experiments and
presents an analysis of the results. Finally, Section 6 concludes this paper and gives direction for further investigations.

2 Interchangeability

The idea behind interchangeability is to make use of values that behave similarly in some or all environments. In addition to other sorts of interchangeability, [Freuder 1991] introduced the concept of interchangeability between two values in the domain of one variable, in either a local or a global environment, if they can be substituted for each other without effecting the environment. Here we briefly explain, with our own words, the main kinds of interchangeabilities in [Freuder 1991] and those we use.

Definition 2.1. Full interchangeability (FI): A value \( a \) in the domain of variable \( V \) is interchangeable with a 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.

The computation of FI may require finding all solutions, and thus is likely to be intractable. [Freuder 1991] also gives a localization of FI, which can be computed in \( O(na^2) \) by considering only constraints incident to the variable:

Definition 2.2. Neighborhood interchangeability (NI): A value \( a \) in the domain of variable \( V \) is neighborhood interchangeable (NI) 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: \( \{ x \mid (a,x) \text{ satisfies } C \} = \{ x \mid (b,x) \text{ satisfies } C \} \). NI is a sufficient, but not a necessary condition for FI.

Both FI and NI do not permit variables other than \( V \) in the CSP to change. [Freuder 1991] also proposes to weaken interchangeability by increasing the boundary of change:

Definition 2.3. Partial interchangeability (PI): A value \( a \) in the domain of variable \( V \) is partially interchangeable (PI) with a value \( b \) in the same domain with respect to a boundary of change, \( A \), which is a subset of variables, iff a solution involving \( a \) remains a solution when \( b \) is substituted for \( a \), with possible different values for the variables in \( A \).

Neighborhood Partial Interchangeability (NPI) is a localization of PI, such that only constraints involving the neighborhood of the subset \( A \) are considered [Choueiry and Noubir 1998]. As such, NPI is a sufficient, but not necessary condition for PI. [Haselböck 1993] had used an extreme instance of this localization, which we call NIC:
Definition 2.4. Neighborhood interchangeability according a Constraint (NIC): A value $a$ in the domain of variable $V_i$ is neighborhood interchangeable across a constraint (NIC) with a value $b$ in the same domain iff $a$ and $b$ are consistent with the same values in another variable $V_j$ according to one constraint, $C$. NIC is a sufficient condition of NPI.

Once interchangeable values in a variable are detected, they can be replaced by one representative of the bundle, thus reducing the size of the initial problem. Further, [Freuder 1991] noted that interchangeable sets can be computed either before search, or interleaved with the instantiation of variables during search. When interchangeable sets are computed during search, they constitute dynamic interchangeability.

Definition 2.5. Dynamic NPI (DNPI): We define DNPI as the NPI for a variable $V$ with the boundary of change $A$ as the union of the past variables (those already instantiated) and the current variable, $V$.

3 Search strategies

Below we review five search strategies we use as our basis, see Figure 1: forward checking with static least-domain (FC-BT-sld), forward checking with dynamic least-domain (FC-BT-dld), forward checking with dynamic variable-value ordering according to promise (FC-BT-promise), forward checking with static (FC-NIC-sld) and dynamic (FC-DNPI-sld) bundling.

![Figure 1: Basic search algorithms.](image)

3.1 Forward checking

In the forward-checking (FC-BT) search algorithm [Haralick and Elliott 1980], a search tree is generated by sequentially instantiating variables of the problem.
solution to the CSP is a path of length equal to the number of variables. When a variable is instantiated (current variable), the domains of uninstantiated variables (future variables) are filtered to be consistent with the current instantiation. As a result, the domains of future variables are maintained always consistent with the instantiations of the current and past variables along a given path. The first strategy that we consider is FC-BT with a static least-domain (sld) ordering of the variables. Static least-domain consists of sorting the variables in an increasing order of their domain size before search is started and instantiating the variables in this order during search.

3.2 Dynamic variable ordering

Dynamic variable ordering is an adaptation of FC-BT in which the order of the uninstantiated variables is reconsidered during search. As a result, two different paths in the search tree may exhibit two different sequences of instantiated variables. [Bacchus and van Run 1995] showed that a dynamic variable ordering generally yields a more effective traversal of the search space, and therefore a faster search, than static variable ordering. The variable ordering heuristic we choose in our analysis is dynamic least-domain (dld). Note that this is the same as the method of [Bacchus and van Run 1995], called MRV.

**Definition 3.1.** Dynamic least-domain (dld): Dynamic least-domain is a dynamic ordering of the variables in which, at each step during search, the variable with the smallest remaining domain is chosen to be instantiated.

**Proposition 3.2.** All the nodes in a tree of a dld search that have the same parent necessarily pertain to the same variable.

The proof of this is trivial and is based on the following observation, as we illustrate in Figure 2. After choosing \((V_1, a)\) at depth level \(h\), we choose next the variable that has the smallest current domain among all uninstantiated variables.
Let $V_2$ be such a variable, and will yield a child to $(V_1, a)$ at level $h + 1$ in the tree. Since values are only removed from the domain during the instantiation of a variable, the size of the domain of $V_2$ necessarily shrinks. This guarantees that the same variable $V_2$ will be chosen for the following instantiation of the next child of $(V_1, a)$.

### 3.3 Dynamic variable-value ordering

Some techniques for dynamic variable ordering also include dynamic value ordering [Keng and Yun 1989; Geelen 1992]. In these cases, a heuristic considers all possible values in all future variables looking for the best variable-value pair to instantiate.

An interesting phenomenon occurs in dynamic variable-value ordering. Unlikeddl, two nodes in the tree that have the same parent do not necessarily pertain to the same variable as illustrated in Figure 3. Indeed, at any one particular level of the search tree, variable-value pairs pertaining to different variables may be chosen to be visited.

![Figure 3: A possible search tree with dynamic variable-value ordering.](image)

In Figure 3, a search tree with dynamic variable-value ordering is shown. Suppose that at level $h$, we decide to expand the node $(V_1, a)$. When we are considering a sibling for $(V_1, a)$, a search with static value ordering can only consider another value in $V_1$, such as $(V_1, b)$. However, a search with dynamic value ordering may choose an entirely new variable-value pair (here, $(V_3, b)$). Notice that along this new path, $V_1$ has not yet been instantiated and will appear as a future variable. Importantly, the domain of $V_1$ along this new path must not include $a$. This is because all possibilities with $(V_1, a)$ have already been considered in the subtree rooted at $(V_1, a)$ at level $h$. When the search backtracks to level $(h - 1)$, the value $a$ must be returned to the domain of $V_1$, but not before. While this shows that a variable may be both in the past and in the future, no variable can appear in
the past and the future along *any one path* in the tree—this would imply that the variable has more than one assignment.

In our analysis we use the *promise* heuristic of [Geelen 1992] for dynamic variable-value ordering.

**Definition 3.3.** Promise dynamic variable-value ordering: *Promise is a dynamic variable-value ordering in which, at the instantiation of every variable, every possible value for every uninstantiated variable is considered. The variable-value pair chosen is the one that leaves the largest number of remaining possible solutions. Promise returns, as a fortunate side effect, the domains of future variables as if they were filtered by forward checking.*

### 3.4 Static bundling with FC-NIC

[Haselböck 1993] proposes to compute all NIC sets (for all variables according to every constraint) as a preprocessing step prior to search then use these static interchangeabilities during search. For any given variable $V$, the constraints on $V$ are divided into two groups. *Past* constraints are constraints between $V$ and any variable already instantiated (in the past), and *future* constraints are those constraints between $V$ and any variable not yet instantiated (in the future). In FC-NIC, all NIC sets of $V$ are computed prior to search. $V$ has at most $(n - 1)$ NIC sets, one for each constraint incident to $V$.

The sets computed according to past constraints are used to the revise the domain of $V$. When $V$ is considered for instantiation, the domain of $V$ is bundled by taking the intersection of the NIC sets of $V$ across future constraints. To instantiate $V$, we choose one of these bundles and assign it to $V$. In turn, the domains of future variables are revised, using their respective NIC sets, to be consistent with the particular bundle assigned to $V$. A detailed explanation of this search can be found in [Haselböck 1993].

### 3.5 Dynamic bundling with FC-DNPI

As noted by [Freuder 1991], interchangeability sets can be re-computed after some instantiations are made in the course of backtrack search. Because instantiations restrict the domain of the instantiated variables to the assigned values, interchangeabilities that did not exist before search began may present themselves during search. This dynamic interchangeability must obviously be computed in steps interleaved with search.

In a companion paper [Choueiry and Beckwith 2001], we present a search procedure called FC-DNPI, which we briefly describe here. In FC-DNPI, for a current variable $V$, NPI is calculated with the boundary of change $A = \{V$ and
every variable in the past}. This NPI is calculated using the joint discrimination tree (JDT) introduced in [Choueiry and Noubir 1998]. (As we argue in [Choueiry and Beckwith 2001], this JDT can be also exploited for forward checking.) The JDT partitions the domain of $V$ into bundles, and one of these bundles is chosen (either randomly, or by a heuristic in dynamic value ordering) to be assigned to $V$. In [Choueiry and Beckwith 2001], we prove that this mechanism is always worthwhile when searching to find all solutions.

4 New hybrid algorithms

Starting from these five basic algorithms, we generate four hybrid algorithms by combining various ordering heuristics with various bundling strategies, as shown in Figure 4. Our five base algorithms build on the pseudo-code for forward checking (FC-BT) given in [Prosser 1993] and implement exactly one of the following strategies. **Ordering:** static variable-value ordering, dynamic-variable/static-value ordering, and dynamic variable-value ordering. **Bundling:** non-bundling backtrack search, static bundling, and dynamic bundling. We introduce four new search algorithms that combine one ordering and one bundling strategy of the one listed above.

We assume familiarity FC-BT-sld [Haralick and Elliott 1980], FC-BT-dld [Bacchus and van Run 1995] and FC-BT-promise [Geelen 1992]. Below, we describe, as pseudo-code, the enhancements needed to generate the new dynamic-ordering algorithms (i.e., FC-NIC-dld, FC-NIC-promise, FC-DNPI-dld and FC-DNPI-promise) starting from their respective static-ordering procedures (i.e., FC-NIC-sld and FC-DNPI-sld).

To modify a strategy from a static ordering to a dynamic ordering, we introduce a new function \( \text{NextVar} \). \( \text{NextVar} \) takes as input the lists of future variables and that of past variables (needed to find the boundary of change in DNPI), and returns

![Figure 4: Search Algorithms.](image-url)
a choice for the next expansion. For *static* orderings, NextVar merely pops the first variable from the list of future variables sorted in increasing domain size. For *dynamic* orderings, we specialize NextVar in three ways: NextVar-dld, NextVar-NIC-promise, and NextVar-DNPI-promise as shown below.

<table>
<thead>
<tr>
<th>Search</th>
<th>FC-NIC-dld and FC-DNPI-dld</th>
<th>FC-NIC-promise</th>
<th>FC-DNPI-promise</th>
</tr>
</thead>
<tbody>
<tr>
<td>NextVar</td>
<td>NextVar-dld</td>
<td>NextVar-NIC-promise</td>
<td>NextVar-DNPI-promise</td>
</tr>
<tr>
<td>Pseudocode</td>
<td>Figure 5</td>
<td>Figure 6</td>
<td>Figure 7</td>
</tr>
<tr>
<td>Output</td>
<td>The next variable</td>
<td>The next variable-value pair and information for forward checking</td>
<td></td>
</tr>
</tbody>
</table>

As specified above, NextVar-dld returns the choice for the next variable according to the heuristic in place—in our case, the variable whose domain has the least number of values. NextVar-NIC-promise and NextVar-DNPI-promise return the next variable-value pair (where a value is a bundle) and the filtered domains for each of the corresponding future variables.

To understand why NextVar with promise returns the future variables with filtered domains, recall that in both FC-BT-promise and FC-DNPI, forward-checking is performed implicitly. With promise, the remaining problem size for each possible value (or bundle) in each possible variable is calculated, and the most-promising value in the least-promising variable is chosen. Similarly for FC-DNPI, the JDT for a given variable provides all the future variables and their remaining domains. Both promise and the JDT implicitly ‘compute’ forward checking. Therefore, when a variable-value pair is chosen, forward checking need not be executed.
NextVar-dld (Future-Vars, Past-Vars):

Begin
    best-var ← nil
    least-domain ← 0

    /* choose the variable with smallest domain */
    For each variable $V_i$ in Future-Vars
        if $V_i$ domain has fewer elements than least-domain
            best-var ← $V_i$
            least-domain ← number of elements in domain of $V_i$
    End

End

Figure 5: Finding the next variable to expand using dld.
NextVar-NIC-promise($Future-Vars$, $Past-Vars$):

Begin
best-var $\leftarrow$ nil
best-bundle $\leftarrow$ nil
min-var-promise $\leftarrow$ big-number
/* choose the variable with minimum promise */
For each variable $V_i$ in $Future-Vars$
  promise-var $\leftarrow$ 0
  past-constraints $\leftarrow$ all constraints between $V_i$
  and any variable in $Past-Vars$
  future-constraints $\leftarrow$ all constraints between $V_i$
  and any variable in $Future-Vars$
Partition domain of $V_i$ according to NIC on intersection of
  all future-constraints
max-bundle-promise $\leftarrow$ 0
local-best-bundle $\leftarrow$ nil
/* choose the bundle with the maximum promise */
For each bundle $b$ in $V_i$.
  promise-bundle $\leftarrow$ 1
  For each variable $V_j$ in path of JDT
    left $\leftarrow$ domain remaining for $V_j$
    promise-bundle $\leftarrow$ promise-bundle $\times$ left
    if (promise-bundle $>$ max-bundle-promise)
      local-best-bundle $\leftarrow$ $b$
      promise-var $\leftarrow$ promise-var + promise-bundle
      if (promise-var $<$ min-promise-var)
        best-var $\leftarrow$ $V_i$
        best-bundle $\leftarrow$ local-best-bundle
  return best-var, best-bundle, and $Future-Vars$
End

Figure 6: Finding the next variable to expand using promise in FC-NIC.
NextVar-DNPI-promise(Future-Vars, Past-Vars):

Begin

best-var ← nil
best-bundle ← nil
min-var-promise ← big-number
/* choose the variable with minimum promise*/
For each variable $V_i$ in Future-Vars

promise-var ← 0
Boundary of change $S ← V_i \cup Past-Vars$
Partition domain of $V_i$ according to NPI according to $S$. /* Now, each bundle has an associated JDT */
max-bundle-promise ← 0
local-best-bundle ← nil
/* choose the bundle with the maximum promise */
For each bundle $b$ in DNPI partition of $V_i$.  
promise-bundle ← 1
For each variable $V_j$ in path of JDT

left ← domain remaining for $V_j$
promise-bundle ← promise-bundle \times left
if (promise-bundle > max-bundle-promise)
local-best-bundle ← b
promise-var ← promise-var + promise-bundle
if (promise-var < min-promise-var)
best-var ← $V_i$
best-bundle ← local-best-bundle

return best-var, best-bundle, and Future-Vars

End

Figure 7: Finding the next variable to expand using promise in FC-DNPI.

Each search calls its own NextVar function, tailored for that particular search. It then uses the information returned to proceed with search. As we will see in the next section, the searches that use bundling indeed end up with a smaller search space, yielding a more effective search.
5 Experiments

We performed tests on a battery of random problems generated using the problem generator of [Bacchus and van Run 1995]. This generator creates random problems of a specified number of variables \( (n) \), domain size \( (\alpha) \), constraint tightness \( (t) \) and density \( (d) \). It does not intentionally introduce nor control interchangeability in the problem, which allows us to test our algorithm in the least advantageous conditions. We experimented on instances of 10 variables \( (n = 10) \), fixed domain size of 5 \( (\alpha = 5) \), constraint density \( d = \{.1, .5, .9\} \), and constraint tightness \( t = \{.04, .12, \ldots, .92\} \) with a step of .08. We generated 20 random instances for each density and tightness, for a total pool of 720 random problems. The measured information consists of the total CPU time and the number of solution bundles. For each measurement point, the results were averaged over the 20 instances. In order to reduce the duration of our experiments to a reasonable value, we chose to make all problems arc-consistent (AC-3) before search began. Since this is done uniformly in all experiments and for all strategies, it does not affect the quality of our conclusions. We ran each of the nine searches of Figure 4 on every instance to find all solutions.

An anomaly of random problems is that problems with dense, tight constraints are likely to have no solutions. Therefore we supplement the random problems with puzzles—where the constraints are dense and tight, but the problem is contrived to have one or more solutions. We include in our problem set instances of \( N \)-queens with \( n = \{3, 4, \ldots, 8\} \) as well as three versions of the Zebra problem. The first zebra, Zebra-1, is the traditional zebra problem, with one possible solution, specified in [Régis 1994]. The second, Zebra-11, is Prosser’s Zebra problem [Prosser 1993] and has 11 solutions. Finally, the last zebra instance is Zebra-210 is created from Zebra-1 by removing the two unary constraints on the variables MILK and NORWEGIAN and has 210 solutions. Below we describe our results and analyze them.

5.1 Results and analysis

Before we discuss our results in detail, it is important to note that:

- Our compiled code has not been optimized for run time and the resolution of the clock is of 10 ms. Fractions are due to averaging. Thus, all reported CPU times should be considered as relative measures.

- In the time values reported, we include the time necessary to detect all interchangeabilities. That is, for FC-NIC, computing NIC sets before search; and for FC-DNPI, computing the DNPI sets repeatedly during search.
• For random problems with \( t > .44 \), most instances were found insolvable because they were not arc-consistent even before starting search. Therefore the time to find this absence of solution quickly falls to the time to perform arc-consistency—less than 10 milliseconds on these problems. We omit these data from the charts to save space and avoid cluttering.

From observing the entries and charts in Table 1 and Table 2, we can summarize our observations as follows:

1. **Promise** is not a good ordering heuristic for finding all the solutions to a CSP. Indeed, its performance is always poor, especially in terms of CPU-time. This is true both in general and also when compared with any non-promise based \( \text{sld} \) or \( \text{dld} \) strategy. This holds for non-bundling, static bundling and dynamic bundling. The problem with *promise* is the large number of constraint checks as shown in both tables. Although *promise* seems to often cut the number of nodes visited, when \( t < 0.28 \) (Table 1) it seems to lose its advantage and justification.

2. Bundling is worthwhile. This is obvious especially in Table 1, where we see that FC-BT searches are consistently beaten on all criteria by both FC-NIC searches and FC-DNPI searches. Further, dynamic bundling (FC-DNPI searches) is always better than static bundling (FC-NIC searches) in terms of Nodes Visited (NV), Constraint Checks (CC) and Solution Bundles (SB), and almost always better than the FC-NIC searches in terms of CPU time. This holds for both static and dynamic bundling, and supports the claims made in our companion paper [Choueiry and Beckwith 2001].

3. Dynamic ordering (\( \text{dld} \)) is almost always better than static ordering (\( \text{sld} \)).

4. The improvement made by dynamic bundling is bigger than the improvement by dynamic ordering, though less consistently.

6 Conclusion and future work

Ordering strategies and bundling mechanisms are orthogonal processes for improving the performance of search. The former allows a better navigation in the search space and the latter shrinks its size. We demonstrate that both are successful in making search run faster, and we propose a combination that we prove empirically to be worthwhile. We also provide support to the intuition we state in [Choueiry and Beckwith 2001] that dynamic ordering always improves the quality of the
<table>
<thead>
<tr>
<th>t \ d</th>
<th>Nodes Visited NV</th>
<th>Constraint Checks CC</th>
<th>Solution Bundles SB</th>
<th>Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.04</td>
<td>FC-BT sld 7356600 4325913 2573138</td>
<td>270814 1 3978052 3364460</td>
<td>5710371 3256031 1859533</td>
<td>114577 122483 93760</td>
</tr>
<tr>
<td></td>
<td>did 7162909 4107646 2369881</td>
<td>2312579 3010105 2946881</td>
<td>5710371 3256031 1859533</td>
<td>106967 38542 86431</td>
</tr>
<tr>
<td></td>
<td>promise 7462979 4419953 2687003</td>
<td>2797807 1849853 1153386</td>
<td>5710371 3256031 1859533</td>
<td>604599 3880080 2478908</td>
</tr>
<tr>
<td>0.12</td>
<td>FC-NIC sld 2530 31813 124288</td>
<td>4320 35799 289700</td>
<td>624 9177 46405</td>
<td>289 4044 21760</td>
</tr>
<tr>
<td></td>
<td>did 1930 24811 78900</td>
<td>3755 46765 173030</td>
<td>474 7546 26665</td>
<td>232 3358 12628</td>
</tr>
<tr>
<td></td>
<td>promise 3728 62426 155365</td>
<td>53885 992051 2497716</td>
<td>1310 22404 55570</td>
<td>2261 39204 98276</td>
</tr>
<tr>
<td>0.20</td>
<td>FC-DNPI sld 2038 20909 77460</td>
<td>12043 161051 683584</td>
<td>481 5353 21378</td>
<td>440 5378 23239</td>
</tr>
<tr>
<td></td>
<td>did 1831 24034 69037</td>
<td>11334 197127 642579</td>
<td>450 7336 23098</td>
<td>398 6666 21760</td>
</tr>
<tr>
<td></td>
<td>promise 3376 54702 107864</td>
<td>143230 1980748 3759975</td>
<td>1116 16786 29931</td>
<td>4840 71125 135478</td>
</tr>
<tr>
<td>0.28</td>
<td>FC-BT sld 2638985 516245 106796</td>
<td>1310551 539847 214263</td>
<td>1890654 325481 53049</td>
<td>33180 24682 73298</td>
</tr>
<tr>
<td></td>
<td>did 2401250 341445 78143</td>
<td>997743 237027 137380</td>
<td>1890654 325481 53049</td>
<td>41438 38363 41661</td>
</tr>
<tr>
<td></td>
<td>promise 216860 521830 107606</td>
<td>1033101 2214144 6711414</td>
<td>1890654 325481 53049</td>
<td>216265 51397 130028</td>
</tr>
<tr>
<td>0.44</td>
<td>FC-NIC sld 30701 10399 61365</td>
<td>46913 213925 152561</td>
<td>9166 36268 18875</td>
<td>2842 13458 9533</td>
</tr>
<tr>
<td></td>
<td>did 19591 61885 29400</td>
<td>37415 143665 87262</td>
<td>6039 21553 10436</td>
<td>2026 8068 4931</td>
</tr>
<tr>
<td></td>
<td>promise 98362 226774 65725</td>
<td>1898884 4539830 1423742</td>
<td>40358 87655 23267</td>
<td>62678 152548 92223</td>
</tr>
<tr>
<td>0.86</td>
<td>FC-DNPI sld 19835 61197 35587</td>
<td>63797 239995 163366</td>
<td>5607 17773 9301</td>
<td>2621 9929 6936</td>
</tr>
<tr>
<td></td>
<td>did 17926 49903 23612</td>
<td>65550 224590 117149</td>
<td>5475 17131 7989</td>
<td>2626 8716 4760</td>
</tr>
<tr>
<td></td>
<td>promise 68089 120095 26394</td>
<td>1869381 4152037 1139424</td>
<td>26797 38852 6511</td>
<td>68681 140900 38804</td>
</tr>
</tbody>
</table>

Table 1: Results on random problems.
Table 2: Comparison of CPU-Time (top) and Solution Bundling (bottom) for four searches.

<table>
<thead>
<tr>
<th>Search</th>
<th>Ordering</th>
<th>NV</th>
<th>CC</th>
<th>Time</th>
<th>SB</th>
</tr>
</thead>
<tbody>
<tr>
<td>FC</td>
<td>sld</td>
<td>2186</td>
<td>15508</td>
<td>290</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>dld</td>
<td>1215</td>
<td>10243</td>
<td>200</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>promise</td>
<td>869</td>
<td>391982</td>
<td>3150</td>
<td>0</td>
</tr>
<tr>
<td>FC-NIC</td>
<td>sld</td>
<td>2186</td>
<td>22196</td>
<td>1020</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>dld</td>
<td>1209</td>
<td>16708</td>
<td>570</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>promise</td>
<td>923</td>
<td>190901</td>
<td>3300</td>
<td>92</td>
</tr>
<tr>
<td>FC-DNPI</td>
<td>sld</td>
<td>2134</td>
<td>22196</td>
<td>1020</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>dld</td>
<td>1216</td>
<td>10356</td>
<td>290</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>promise</td>
<td>824</td>
<td>177526</td>
<td>3520</td>
<td>92</td>
</tr>
</tbody>
</table>

Table 2: Puzzle data.
bundling, and almost always improved time. In the future, we intend to pursue the following directions:

- Investigate the effects of dynamic bundling for finding a single solution bundle and check whether the performance of \texttt{promise} can regain relatively to other ordering heuristics.

- Create a random generator such as the one described in [Freuder and Sabin 1995], and test these methods on problems with various degrees of interchangeability.

- Demonstrate that dynamic bundling remains competitive when integrated to search strategies that are based on maintaining arc-consistency (MAC).

- Report our work on non-binary CSPs.

\textbf{Acknowledgements}

Thanks to Robert Glaubius for formatting data into Excel charts and \LaTeX\ tables.

\textbf{References}


