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

Amy M. Beckwith and Berthe Y. Choueiry
Constraint Systems Laboratory
Department of Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln NE 68588-0115
Email: {abeckwit | choueiry }@cse.unl.edu

Technical Report CSL-01-03

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 it 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

List of Tables

List of Figures

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 `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 `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 verse.*

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.

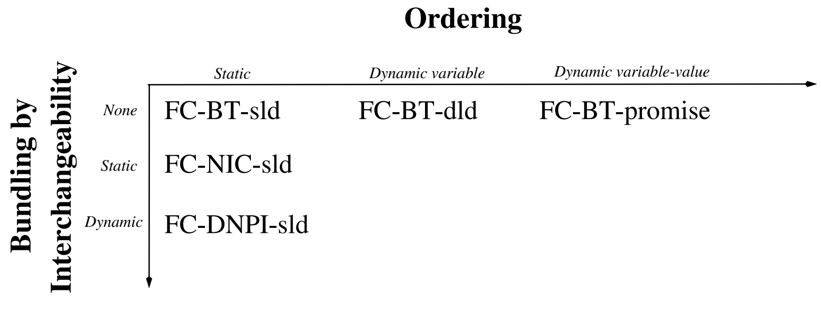


Figure 1: *Basic search algorithms.*

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. A 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 adaption 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 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.

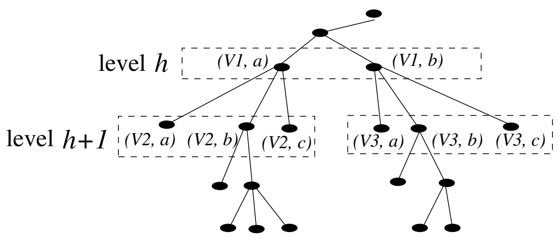


Figure 2: *Example of a* `dld` *search tree.*

The proof of this is trivial and is based on the following observation, as we illustrate in Figure 2. After choosing $(V1, a)$ at depth level h, we choose next the variable that has the smallest current domain among all uninstantiated variables.

Let $V2$ be such a variable, and will yield a child to $(V1, a)$ 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 $V2$ necessarily shrinks. This guarantees that the same variable $V2$ will be chosen for the following instantiation of the next child of $(V1, a)$.

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. Unlike `dld`, 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.

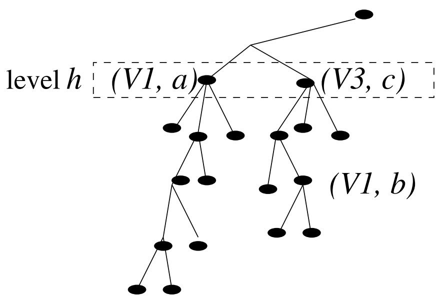


Figure 3: *A possible search tree with dynamic variable-value ordering.*

In Figure 3, a search tree with dynamic variable-value ordering is shown. Suppose that at level h, we decide to expand the node $(V1, a)$. When we are considering a sibling for $(V1, a)$, a search with static *value* ordering can only consider another value in $V1$, such as $(V1, b)$. However, a search with dynamic value ordering may choose an entirely new variable-value pair (here, $(V3, b)$). Notice that along this new path, $V1$ has not yet been instantiated and will appear as a future variable. Importantly, the domain of $V1$ along this new path must not include a. This is because all possibilities with $(V1, a)$ have already been considered in the subtree rooted at $(V1, a)$ at level h. When the search backtracks to level $(h - 1)$, the value a must be returned to the domain of $V1$, 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 check-

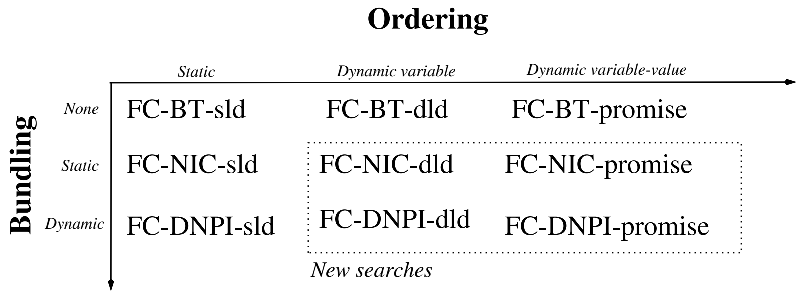


Figure 4: *Search Algorithms.*

ing (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 `NextVar`. `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 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.

Search	FC-NIC-dld and FC-DNPI-dld	FC-NIC-promise	FC-DNPI-promise
`NextVar`	`NextVar-dld`	`NextVar-NIC-promise`	`NextVar-DNPI-promise`
Pseudocode	Figure 5	Figure 6	Figure 7
Output	The next variable	The next variable-value pair and information for forward checking	

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 $\leftarrow$ nil
least-domain $\leftarrow$ 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 $\leftarrow V_i$
        least-domain $\leftarrow$ number of elements in domain of V_i
return *best-var*
End

Figure 5: *Finding the next variable to expand using* `dld`*.*

```
NextVar-NIC-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
    past-constraints ← all constraints between V_i
            and any variable in Past-Vars
    future-constraints ← 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 ← 0
    local-best-bundle ← nil
    /* choose the bundle with the maximum promise */
    For each bundle b in V_i.
          promise-bundle ← 1
          For each variable V_j in path of JDT
                left ← domain remaining for V_j
                promise-bundle ← promise-bundle ×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 6: *Finding the next variable to expand using* `promise` *in FC-NIC.*

NextVar-DNPI-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
    Boundary of change $S \leftarrow V_i \cup$ *Past-Vars*
    Partition domain of V_i according to NPI according to S.
    /* Now, each bundle has an associated JDT */
    max-bundle-promise $\leftarrow$ 0
    local-best-bundle $\leftarrow$ nil
    /* choose the bundle with the maximum promise */
    For each bundle b in DNPI partition of 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 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 (a), 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 (a = 5), constraint density $d = \{.1, .5, .9\}$, and constraint tightness $t = \{.04, 0.12, \ldots, .92\}$ 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 [Régin 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 `sld` or `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 justication.

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 (`dld`) is almost always better than static ordering (`sld`).

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

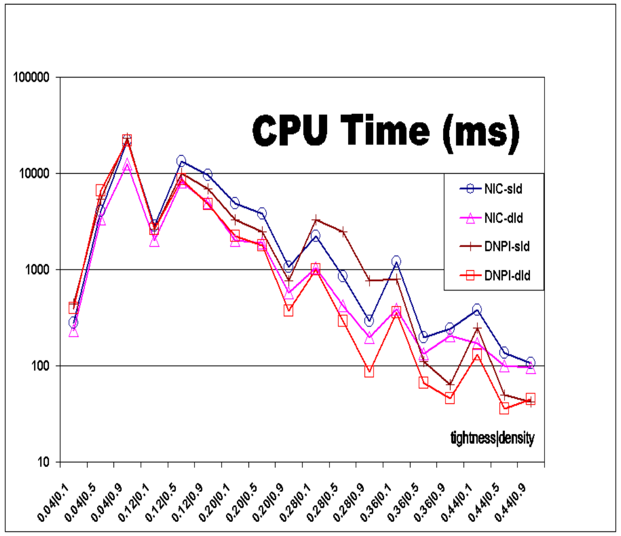
6 Conclusion and future work

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

$t \setminus d$			Nodes Visited NV			Constraint Checks CC			Solution Bundles SB			Time [ms]		
			0.1	**0.5**	**0.9**	**0.1**	**0.5**	**0.9**	**0.1**	**0.5**	**0.9**	**0.1**	**0.5**	**0.9**
0.04	FC-BT	sld	7356600	4325913	2573138	2708141	3978502	3364460	5710371	3256031	1859533	114577	122483	93760
		dld	7162999	4107646	2369881	2312579	3101095	2946881	5710371	3256031	1859533	106967	98542	86431
		promise	7462692	4419965	2687086	279786157	184895153	115331018	5710371	3256031	1859533	6045936	3880060	2478908
	FC-NIC	sld	2530	31813	142848	4320	55799	289700	624	9177	46403	280	4044	21760
		dld	1930	24811	78900	3755	46765	173030	474	7546	26665	232	3358	12628
		promise	3728	62426	155365	53885	992051	2497716	1310	22404	55570	2261	39204	98276
	FC-DNPI	sld	2038	20098	77460	12043	161051	685384	481	5353	23178	440	5378	23239
		dld	1831	24034	69037	11334	197127	642579	450	7336	23098	398	6666	21760
		promise	3376	54702	107864	143230	1980748	3759975	1116	16786	29931	4840	71125	135478
0.12	FC-BT	sld	2638985	516245	106796	1310551	539847	214263	1890654	324581	53049	52198	17652	6389
		dld	2401250	434145	78143	997743	427077	137380	1890654	324581	53049	41438	13863	4166
		promise	2641860	521830	107606	103105050	25266114	6371414	1890654	324581	53049	2162975	513907	130028
	FC-NIC	sld	30701	110709	61365	46913	219258	152561	9166	36268	18875	2842	13458	9533
		dld	19591	61885	29400	37415	143665	87262	6039	21553	10436	2026	8068	4931
		promise	98362	226774	65725	1898884	4539850	1423742	40358	87655	23267	62678	152548	49223
	FC-DNPI	sld	19835	61197	35587	63797	249995	163366	5607	17773	9301	2621	9929	6936
		dld	17926	49903	23612	65550	224590	117149	5475	17131	7989	2626	8716	4760
		promise	68908	120095	26394	1869381	4152037	1139424	26797	38852	6511	66811	146900	38804
0.20	FC-BT	sld	828805	56534	4593	561741	90370	17609	520957	25361	1138	21904	2957	580
		dld	673830	38434	2442	294998	52117	8726	520957	25361	1138	12208	1520	294
		promise	819046	53895	3702	36397138	3973663	579505	520957	25361	1138	728644	77246	11054
	FC-NIC	sld	47531	29802	4143	95700	65784	20156	14278	8240	752	4892	3796	1062
		dld	19341	13206	1708	40322	39568	11850	5903	4433	450	2004	1893	573
		promise	147343	38546	3204	3194075	1026862	168891	59656	13211	826	100052	32347	5016
	FC-DNPI	sld	28049	16971	3193	77275	57899	16781	7895	3942	467	3316	2456	766
		dld	18279	11167	1513	52768	41371	8483	5652	3599	374	2248	1762	370
		promise	97874	23072	1938	2788417	1059954	178154	37028	6190	346	94764	34698	5408
0.28	FC-BT	sld	230354	5926	372	156328	14278	3014	130375	1560	19	6388	459	108
		dld	174763	2931	136	91718	6344	1497	130375	1560	19	3686	200	76
		promise	241875	4826	137	13629920	759178	92807	130375	1560	19	264552	13740	1742
	FC-NIC	sld	23425	4558	369	42816	15668	6929	5848	788	16	2240	850	294
		dld	9393	1551	129	24228	8398	5501	2820	409	11	1038	418	198
		promise	96066	4005	133	2299165	223841	46048	36775	1072	16	68666	6224	1289
	FC-DNPI	sld	15078	3210	333	36797	12284	2975	3514	472	11	1652	560	148
		dld	8343	1370	126	23529	5819	1509	2480	339	9	995	293	87
		promise	57032	2954	111	1782813	249061	50094	20093	597	9	57218	7694	1496
0.36	FC-BT	sld	73610	535	68	67515	2587	839	33493	50	0	2617	90	45
		dld	45584	207	29	23608	1210	540	33493	50	0	1012	52	38
		promise	72097	308	16	5013315	138124	26606	33493	50	0	93314	2398	608
	FC-NIC	sld	11637	488	72	22871	4998	4711	2879	30	0	1204	196	243
		dld	3392	171	29	9144	3693	4409	940	22	0	390	132	206
		promise	26210	253	16	692436	56570	22430	9167	37	0	20253	1410	582
	FC-DNPI	sld	7670	432	66	18888	2504	838	1698	23	0	802	111	64
		dld	3223	170	29	8344	1206	543	880	21	0	359	66	46
		promise	10443	286	17	368990	68995	21878	3298	26	0	11369	1935	586
0.44	FC-BT	sld	17784	136	12	16665	918	167	6030	4	0	691	46	36
		dld	8687	38	6	5839	400	121	6030	4	0	251	26	32
		promise	15882	51	3	1617320	40482	5329	6030	4	0	28539	682	142
	FC-NIC	sld	3283	132	13	7121	3256	1804	614	2	0	380	137	106
		dld	1013	36	7	3875	2750	1662	252	2	0	171	100	96
		promise	7132	47	2	265454	24077	4998	2254	3	0	7077	634	239
	FC-DNPI	sld	2160	119	13	5077	898	178	374	2	0	245	50	42
		dld	940	39	6	2795	417	115	230	2	0	131	36	45
		promise	1996	286	3	90665	27266	4751	529	3	0	2670	758	166

Table 1: *Results on random problems.*

Table 2: *Plots for random problems and table for puzzles.*



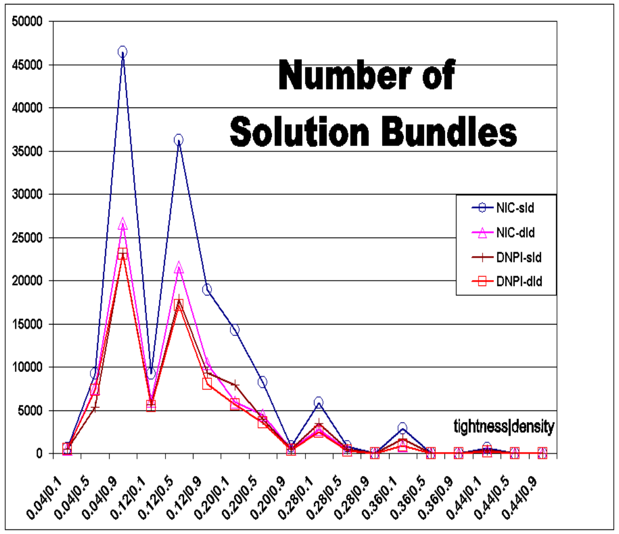


Table 2: *Comparison of CPU-Time (top) and Solution Bundling (bottom) for four searches.*

	Search	Ordering	NV	CC	Time	SB
8-Queens	FC	sld	2186	15508	290	0
		dld	1215	10243	200	0
		promise	869	391982	3150	0
	FC-NIC	sld	2186	22196	1020	92
		dld	1209	16708	570	92
		promise	923	190901	3300	92
	FC-DNPI	sld	2134	15508	540	92
		dld	1216	10356	290	92
		promise	824	177526	3520	92
Zebra	FC	sld	209	972	30	0
		dld	81	522	30	0
	FC-NIC	sld	209	4798	190	1
		dld	81	3612	70	1
		promise	59	56535	1130	1
	FC-DNPI	sld	175	972	40	1
		dld	79	522	30	1
		promise	92	60915	1450	1
Zebra-2	FC	sld	922	4101	110	0
		dld	377	2133	50	0
	FC-NIC	sld	922	9527	390	11
		dld	359	5216	180	11
		promise	302	206213	3950	11
	FC-DNPI	sld	809	4101	200	11
		dld	363	2129	100	11
		promise	333	163690	3700	11
Zebra-3	FC	sld	285668	1803980	47050	0
		dld	4754	22287	670	0
	FC-NIC	sld	285668	2018342	111690	210
		dld	4754	28937	1240	210
		promise	4969	3368816	67840	210
	FC-DNPI	sld	268812	1803980	51980	210
		dld	4682	22287	800	210
		promise	2725	1032091	23500	210

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 `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.

Acknowledgements

Thanks to Robert Glaubius for formatting data into Excel charts and LaTeXtables.

References

Bacchus, Fahiem and Run, P.van 1995. Dynamic Variable Ordering in CSPs. In *Principles and Practice of Constraint Programming, CP'95. Lecture Notes in Artificial Intelligence 976*. Springer Verlag. 258–275.

Choueiry, Berthe Y. and Beckwith, Amy M. 2001. Techniques for Bundling the Solution Space of Finite Constraint Satisfaction Problems. Technical Report CSL-01-02. `consystlab.unl.edu/CSL-01-02.ps`, University of Nebraska-Lincoln.

Choueiry, Berthe Y. and Noubir, Guevara 1998. On the Computation of Local Interchangeability in Discrete Constraint Satisfaction Problems. In *Proc. of AAAI-98*, Madison, Wisconsin. 326–333. Revised version KSL-98-24, `ksl-web.stanford.edu/KSL_Abstracts/KSL-98-24.html`.

Freuder, Eugene C. and Sabin, Daniel 1995. Interchangeability Supports Abstraction and Reformulation for Constraint Satisfaction. In *Symposium on Abstraction, Reformulation and Approximation, SARA'95*, Ville d'Esterel, Canada. 62–68.

Freuder, Eugene C. 1991. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *Proc. of AAAI-91*, Anaheim, CA. 227–233.

Geelen, Pieter Andreas 1992. Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems. In *Proc. of the 10 th ECAI*, Vienna, Austria. 31–35.

Haralick, Robert M. and Elliott, Gordon L. 1980. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence* 14:263–313.

Haselböck, Alois 1993. Exploiting Interchangeabilities in Constraint Satisfaction Problems. In *Proc. of the 13 th IJCAI*, Chambéry, France. 282–287.

Keng, Naiping and Yun, David Y. Y. 1989. A Planning/Scheduling Methodology for the Constrained Resource Problem. In *Proc. of the 11 th IJCAI*, Detroit, MI. 998–1003.

Prosser, Patrick 1993. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence* 9 (3):268–299.

Régin, Jean-Charles 1994. A filtering algorithm for constraints of difference in constraint satisfaction problems. In *Proc. of AAAI-94*, Seattle, WA. 362–437.