Learned Value-Ordering Heuristics for Constraint Satisfaction

Zhijun Zhang[1] and Susan L. Epstein[1,2]

[1]Computer Science Department, The Graduate Center of the City University of New York
365 Fifth Avenue, New York, NY 10016-4309, USA
zzhang@gc.cuny.edu

[2]Computer Science Department, Hunter College of the City University of New York,
695 Park Avenue, New York, NY 10065, USA
susan.epstein@hunter.cuny.edu

Abstract

In global search for a solution to a constraint satisfaction problem, a value-ordering heuristic predicts which values are most likely to be part of a solution. When such a heuristic uses look-ahead, it often incurs a substantial computational cost. We propose an alternative approach, survivors-first, that gives rise to a family of dynamic value-ordering heuristics that are generic, adaptive, inexpensive to compute, and easy to implement. Survivors-first prefers values that are most often observed to survive propagation during search. This paper explores two algorithms, and several modifications to them, that learn to identify and recommend survivors. Empirical results show that these value-ordering heuristics greatly enhance the performance of several traditional variable-ordering heuristics on a variety of challenging problems.

Introduction

In global search for a solution to a constraint satisfaction problem (*CSP*), variable-ordering and value-ordering heuristics can have substantial impact on search performance. The *best-first* principle for global search advocates the selection of a value most likely to be part of a solution. Value-ordering heuristics typically rely on look-ahead strategies. Strict adherence to the best-first principle, however, often incurs a substantial computational cost as it tries to predict that probability. Our alternative, *survivors-first*, advocates instead the selection of a value most likely to remain an option (i.e., to *survive*) after propagation.

Our thesis is that, instead of look-ahead, promising values can be learned inexpensively during propagation, and then applied by heuristics to speed solution to an individual problem. We investigate here two simple learning methods that support survivors-first, along with effective, adaptive value-ordering heuristics that severely restrict look-ahead at the root of search tree. Our learned value-ordering heuristics are generic, adaptive, inexpensive to compute, and easy to implement. Moreover, search with a variety of traditional variable-ordering heuristics benefits significantly from survivors-first. After fundamental definitions and a brief discussion of related work, this paper describes our learned value-ordering heuristics and initial empirical results. It then introduces modifications that improve their performance and analyzes empirical results on a broad range of challenging problems.

Background and Related Work

Formally, a CSP P is represented by a triple $<X, D, C>$, where X is a finite set of *variables*, D is a set of *domains* individually associated with each variable in X, and C is a set of *constraints*. An *assignment* associates a variable with a value from its domain. A *constraint* c in C consists of a $scope(c) \subseteq X$ and some $relation(c)$ that identifies all acceptable combinations of assignments for variables in $scope(c)$. The variables in *scope(c)* are *neighbors* of one another.

Global search systematically selects an unassigned variable (here, a *focus variable*), and assigns it a value. After value assignment to a focus variable, a *propagation* algorithm infers its impact upon the domains of the *future* (as yet unvalued) variables. Repeated propagation after each value assignment *maintains* consistency among future variables. A *wipeout* occurs when propagation empties the domain of some future variable. After any wipeout, global search *backtracks* (removes some consecutive sequence of previous assignments).

An *instantiation* for a CSP is a set of assignments of values to variables. A *full* instantiation assigns values to all the variables. Global search begins with an empty instantiation and seeks a *solution* (a full instantiation which is acceptable to every constraint in C). A *solvable* CSP has at least one solution; otherwise it is *unsolvable*. This paper

addresses only solvable *binary* CSPs, where each constraint addresses at most two variables.

In general, variable-ordering heuristics drive search to select future variables that are more tightly constrained, while value-ordering heuristics drive search to select values that are more likely to succeed. Nonetheless, most global search algorithms rely mainly on variable-ordering heuristics and propagation. Value-ordering heuristics have been comparatively neglected because they require substantial computational resources in many situations.

Early criteria for value selection estimated the number of solutions to the problem (Dechter and Pearl 1988). Subsequent efforts used ideas from Bayesian networks to improve such estimation (Meisels, Shimony, and Solotorevsky 1997; Vernooy and Harvens 1999; and Kask, Dechter, and Gogate 2004). This work, however, ignored the cost of finding solutions within a subtree of a specific size. Other work used look-ahead strategies to select a value that maximized the product (Geelen 1992; Ginsberg et al. 1990) or the sum of the resulting domain sizes after propagation (Frost and Dechter 1995). When done dynamically (i.e., recomputed after propagation for every assignment), however, all these methods are costly.

More recent work sped solution counting with global constraints (Zanarini and Pesant 2007) or resorted once again to non-adaptive, faster, static value-ordering heuristics, calculated only once, prior to search (Mehta and von Dongen 2005). Nonetheless, even the initial computation to produce a static value ordering is non-trivial for many large problems.

Early work on learning about values focused on *nogoods* (i.e., instantiations that are not part of any solution). Nogoods can guide search away from the re-exploration of fruitless subtrees, particularly with the help of restart (Lecoutre et al. 2007). *Impact*, the domain reduction observed after propagation, can be used heuristically to select both a variable and the value to assign to it (Refalo 2004). More recently, multi-point constructive search has learned *elite solutions*, instantiations most likely to be extended to a solution (Beck 2007). Elite solutions are used in combination with restart to speed search, primarily on optimization problems. Other learning work has identified high-performing value heuristics (Epstein et al. 2005; Minton 1996), but does not tailor the heuristics to an individual problem.

Learned Value-Ordering Heuristics

Let L be an instantiation and let x be a future variable with possible value v and neighboring variable y. A value in the domain of y that is consistent with the assignment of v to x is a *support* for v. If L can be extended to a solution, we say that L is an *extensible instantiation*. Otherwise, L is an

```
revise-3 (variable x_i, constraint c_ij)
    domain-change ← false
    for each v_i in D(x_i)
      if not exist (v_i, v_j) ∈ c_ij then
        remove v_i from D(x_i)
        domain-change ← true
      end if
    end for
    return domain-change
```

(a)

```
revise-3-L (variable x_i, constraint c_ij)
1   domain-change ← false
2   for each v_i in D(x_i)
3     R ← retrieve-R-value(v_i)
4     S ← retrieve-S-value(v_i) + 1
5     unless (v_i, v_j) ∈ c_ij
6       remove v_i from D(x_i)
7       domain-change ← true
8       unless v_i or v_j is the focus variable
9         R ←R + 1
10        update-R-value(v_i, R)
11      end unless
12    end unless
13    update-S-value(v_i, S)
14  end for
15  return domain-change
```

(b)

Figure 1: Pseudocode for (a) the original *revise-3* algorithm (Mackworth 1977) with (b) revisions in boldface to learn which values survive.

inextensible instantiation. If the assignment of v to x changes L from extensible to inextensible, that assignment is a *value mistake* (Harvey 1995). Otherwise, if L remains extensible, v for x is a *compatible assignment* and we say v is *compatible* with L. A *perfect value-ordering heuristic* makes no value mistakes during search for a solution to a solvable CSP.

Search with a perfect value-ordering heuristic would make variable ordering irrelevant, because propagation after a compatible assignment would never cause a wipeout. In general, if L is extensible, the values remaining for the future variables may also be compatible. Propagation after a newly-constructed compatible assignment seeks to remove additional values. Consider an example where values a_1 and a_2 for variable A and values b_1 and b_2 for variable B are all compatible with L, but a_1 is only consistent with b_1 and a_2 is only consistent with b_2. If a perfect value-ordering heuristic selects a_1 for A, then propagation after such an assignment should remove b_2 as an option. In contrast, the compatible value b_1 that *survives* propagation may be part of a solution.

Our approach is assertively optimistic. It assumes that the current instantiation L is extensible, a heuristic to be sure. It also assumes that survival is uniform, that is, that a survivor in one area of the search space has the same probability of survival in other areas. Because constraints are bidirectional (Bessière, Freuder and Régin 1999), supports for a value compatible with L may also be

compatible values. Thus, values that repeatedly survive propagation are likely to be compatible.

Our algorithms observe search and propagation carefully. Rather than look ahead to predict a value most likely to succeed, our algorithms instead learn and use those values that frequently survive propagation during search. They tabulate *R*, how often a value is removed during propagation, and *S* the frequency with which a value is challenged to see if it has support. Our value-ordering heuristics assume that the more frequently a value has been removed, the less likely it is to be compatible, and that the more frequently a value has been challenged, the more likely *R* is to be an accurate predictor of that value's ability to survive.

Our algorithms learn compatible values while enforcing arc consistency. (A binary CSP is said to be *arc consistent* if and only if, for every constraint between pairs of future variables, each value of both variables has at least one support from the domain of its neighbor.) An *AC* algorithm enforces arc consistency; a *MAC* algorithm maintains arc consistency during search (Sabin and Freuder 1994). There are many *AC* algorithms. Any *AC* algorithm, such as *AC-3* (Mackworth 1977) or *AC-2001* (Bessière and Régin 2001; Bessière, Régin, Yap, and Zhang, Y. 2005.), would permit us to monitor the frequency of value removals. *AC-2001* and *AC-3* have the same framework except that *AC-2001* stores the smallest support for each value on each constraint. For simplicity here, we only work within *AC-3* whose maintenance is denoted as *MAC-3*.

Observing *R,* the frequency of value removals, is inexpensive and straightforward. Value removals can be tallied within the key component, *revise-3* of the *AC-3* propagation algorithm, shown in Figure 1(a). *Revise-3* checks every value v of a variable x_i in the scope of a constraint c_{ij} for a support from the other variable x_j constrained by c_{ij}. If no support exists in the domain of x_j, v should be removed from the domain of x_i. In Figure 1(b) lines 3 and 8 – 10 maintain *R*, the removal frequency of each value addressed by *revise-3*. A removal is recorded only when both variables in the scope of c_{ij} are future variables and neither is the current focus variable.

Observing *S,* the frequency with which a value is challenged, is also inexpensive and straightforward. During propagation, a value v of x_i may repeatedly invoke *revise-3* to seek a support from another variable x_j. How often *revise-3* is invoked to seek support for v can be an indication of how difficult it is to remove v. In Figure 1(b), lines 4 and 13 record *S* for each value of each future variable encountered during propagation.

Our first search heuristic, *RVO* (*R*-value-ordering) chooses a value with the lowest recorded *R*. Our second search heuristic, *RSVO* (*R/S*-value-ordering) chooses a value with the lowest observed *R/S* ratio. *RSVO* prefers values that have rarely been removed (small *R*) and frequently challenged (high *S*) during propagation. We use a parameter to define ties. Any domain values whose *R* or *R*/*S* scores differ by no more than *tie-range* (here, 5%) are treated as if they produced a tie. Both heuristics direct search to choose values that have repeatedly survived propagation during search on this problem. The hope is that removal statistics from past propagation closely reflect the likelihood that a value is compatible, and that they can guide search successfully.

Initial Empirical Results

The test bed used here is *ACE* (the Adaptive Constraint Engine) (Epstein, Freuder, and Wallace 2005). ACE is readily configured to do global search with *MAC-3* and particular variable-ordering and value-ordering heuristics. The heuristics used here employ the following definitions. The *domain size* of a variable is the number of values in its dynamic (i.e., after propagation) domain. A variable's *static degree* is the number of its neighbors; its *dynamic degree* is the number of future variables that are its neighbors. A constraint can have a *weight* that is incremented whenever it incurs a wipeout during search. The *weighted degree* of a variable is then the sum of the weight values of all the constraints in whose scope it lies (Boussemart et al. 2004).

We report detailed results on six problem classes, all solvable and binary. The random problems are described with Model B notation, where $\langle n, m, d, t \rangle$ denotes problems on n variables, with maximum domain size m, density d (fraction of the possible pairs of variables that have constraints between them), and tightness t (average fraction of the possible pairs of values *ex*cluded by each constraint) (Gomes et al., 2004). We use problems from <50, 10, 0.38, 0.2> and <50, 10, 0.184, 0.631>. The former is at the phase transition, that is, particularly difficult for its size (n and m values) and type. We also use two classes of solvable balanced 10×10 quasigroups, one with 67 holes (at the phase transition) and one with 74 holes. Finally, composed problems were inspired in part by the hard manufactured problems of (Bayardo Jr. and Schrag, 1996). A *composed* CSP consists of a *central component* joined to one or more *satellites*. The number of variables n, maximum domain size m, density d, and tightness t of the central component and its satellites are specified separately, as are the density and tightness of the links between them. A class of composed graphs is specified here as

$$\langle n, m, d, t \rangle\ s\ \langle n', m', d', t' \rangle\ d''\ t''$$

where the first tuple describes the central component, s is the number of satellites, the second tuple describes the satellite, and the links between the satellite and the central component have density d'' and tightness t''. Satellites are

Table 1: Performance comparison among four CSP solvers equipped with lexical value ordering and variants of learned value-ordering heuristics on 50 problems in <50, 10, 0.38, 0.2>. The modifications *TOP* and *SAC1* and the prefix *m* are described in the next section.

Solver	% Reduction in Checks	% Reduction in CPU time
mdd averaged 29,977,216 checks/problem		
mdd+RVO	19.18%	12.00%
mdd+RVO+TOP	12.25%	2.05%
mdd+RVO+SAC1	61.80%	41.77%
mdd+mRVO	62.58%	51.50%
mdd+RSVO	35.79%	26.50%
mdd+RSVO+TOP	23.80%	10.49%
mdd+RSVO+SAC1	52.67%	35.91%
mdd+mRSVO	65.11%	62.50%
mddd averaged 26,655,702 checks/problem		
mddd+RVO	14.31%	18.36%
mddd+RVO+TOP	7.73%	7.96%
mddd+RVO+ SAC1	50.24%	49.52%
mddd+mRVO	64.10%	64.73%
mddd+RSVO	22.40%	22.71%
mddd+RSVO+TOP	18.40%	16.11%
mddd+RSVO+ SAC1	52.12%	50.25%
mddd+mRSVO	63.50%	63.77%
wdeg averaged 70,441,344 checks/problem		
wdeg+RVO	11.88%	4.20%
wdeg+RVO+TOP	22.95%	6.30%
wdeg+RVO+ SAC1	57.75%	52.65%
wdeg+mRVO	68.68%	64.50%
wdeg+RSVO	28.78%	31.30%
wdeg+RSVO+ TOP	27.49%	26.00%
wdeg+RSVO+ SAC1	67.19%	64.83%
wdeg+mRSVO	68.57%	58.59%
dom/wdeg averaged 26,494,236 checks/problem		
dom/wdeg+RVO	6.42%	8.73%
dom/wdeg+RVO+ TOP	17.76%	26.42%
dom/wdeg+RVO+ SAC1	52.15%	54.11%
dom/wdeg+mRVO	61.69%	58.52%
dom/wdeg+RSVO	6.13%	7.86%
dom/wdeg+RSVO+ TOP	11.73%	15.55%
dom/wdeg+RSVO+ SAC1	53.04%	52.02%
dom/wdeg+mRSVO	63.22%	62.01%

not connected to one another. Some classes of composed problems are particularly difficult for traditional solvers because the "natural" variable ordering is inappropriate, that is, they are better approached with minimum degree than maximum. We test here on two such classes of composed problems.

Table 2: Performance of *dom/wdeg* solvers with lexical value ordering, and with RVO and RSVO value-ordering heuristics on 100 problems from each of two classes of quasigroups. (Although the quasigroup class with 67 holes is generally harder, there was a single problem on 74 holes that proved extremely difficult for *dom/wdeg* and inflated its average performance.)

Solver	Holes	% Reduction in Checks	% Reduction in CPU time
dom/wdeg averaged 121,527 checks/problem			
dom/wdeg+RVO	67	78.32%	65.83%
dom/wdeg+RSVO	67	58.10%	47.34%
dom/wdeg averaged 821,802 checks/problem			
dom/wdeg+RVO	74	89.99%	88.67%
dom/wdeg+RSVO	74	96.26%	94.33%

We test survivors-first with four popular variable-ordering heuristics, one at a time: *mdd* minimizes the ratio of dynamic domain size to static degree, *mddd* minimizes the ratio of dynamic domain size to dynamic degree (Bessière and Régin 1996), *wdeg* prefers a future variable with maximum weighted degree, and *dom*/*wdeg* minimizes the ratio of dynamic domain size to weighted degree (Boussemart et al. 2004). For simplicity, we use *mdd*, *mddd*, *wdeg*, and *dom*/*wdeg* to denote a *basic* MAC-3 solver with the corresponding variable-ordering heuristic and a lexical value-ordering heuristic. Results are reported as reductions in the effort a basic solver required to solve a problem, both in constraint checks and in CPU time. The latter includes the time to learn about value survivors and to apply our value-ordering heuristics. Unless otherwise indicated, all ties are broken by lexical ordering.

Our first set of experiments used the four solvers, to find

Table 3: Performance of *dom/wdeg* solvers with lexical value ordering, and with RVO and RSVO value-ordering heuristics on 100 composed problems from each of two classes.

Solver	% Reduction in Checks	% Reduction in CPU time
<75, 10, 0.216, 0.05, 1, 8, 10, 0.786, 0.55, 0.012, 0.05>		
dom/wdeg averaged 318,057 checks/problem		
dom/wdeg+RVO	16.00%	6.37%
dom/wdeg+RSVO	9.65%	0.32%
<75, 10, 0.216, 0.05, 1, 8, 10, 0.786, 0.55, 0.104, 0.05>		
dom/wdeg averaged 115,875 checks/problem		
dom/wdeg+RVO	8.02%	3.15%
dom/wdeg+RSVO	11.14%	3.94%

the first solution to each of 50 CSPs in <50, 10, 0.38, 0.2>. Table 1 summarizes the performance of each solver alone, and then enhanced in turn by RVO, RSVO, and their modified versions (described in the next section). The results clearly show that RVO and RSVO improve the performance of each of the four variable-ordering heuristics. Our second set of experiments was on the quasigroups. Table 2 reports only on *dom/wdeg* because the other solvers timed out on these problems. Note that *dom/wdeg+RVO* and *dom/wdeg+RSVO* dramatically outperform *dom/wdeg* with lexical value ordering on both problem classes. Our third set of experiments was on the composed CSPs. Again, Table 3 reports only on *dom/wdeg* because the other solvers timed out on these problems.

Modification for Value Mistakes

CSPs in the same class are only ostensibly similar, that is, they may present a broad range of difficultly for any given solution method (Hulubei and O'Sullivan, 2005). Despite their impressive average case performance, *RVO* and *RSVO* may be unsuitable for a specific individual problem. For example, Figure 2 more closely examines the performance of the *dom/wdeg*-based solvers, which showed the least improvement under *RVO* and *RSVO* in Table 1. To construct Figure 2, we first gauged the individual difficulty of our problem set with a *baseline solver* (here, *dom/wdeg*). Then we tabulated our results in ascending order of the difficulty that each solver experienced on each problem, as measured by constraint checks. Clearly, although *dom/wdeg+RVO* and *dom/wdeg+RSVO* do well on problems that are difficult for *dom/wdeg*, their performance is less satisfactory on those that *dom/wdeg* finds easy. The likely explanation is that one or more value mistakes at the top of the search tree

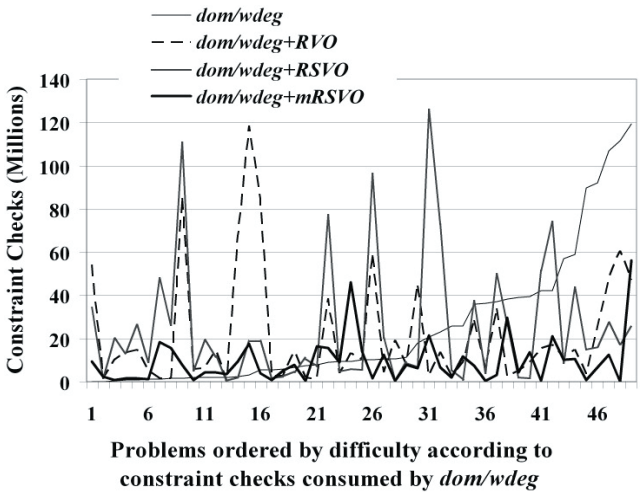

Figure 2: Performance on 50 individual problems by *dom/wdeg* solvers with and without learned value-ordering heuristics on <50, 10, 0.38, 0.2>.

may have misled the solvers into a deep unsolvable subtree. Value mistakes may be frequent near the top of the search tree, where data that supports value ordering is far less informative (Walsh 1997; Meseguer and Walsh 1998).

We therefore make two modifications to decrease the

Table 4: Performance comparison among four CSP solvers equipped with lexical value ordering and variants of learned value-ordering heuristics on 100 problems in <50, 10, 0.184, 0.631>.

	% Reduction in	
Solver	**Checks**	**CPU time**
mdd averaged 1,782,781 checks/problem		
mdd+RVO	2.72%	-1.01%
mdd+RVO+TOP	4.69%	-17.57%
mdd+RVO+SAC1	29.07%	25.98%
mdd+mRVO	24.54%	16.92%
mdd+RSVO	6.36%	0.30%
mdd+RSVO+TOP	-0.59%	-16.78%
mdd+RSVO+SAC1	23.00%	16.82%
mdd+mRSVO	22.74%	17.52%
mddd averaged 1,519,155 checks/problem		
mddd+RVO	15.94%	13.05%
mddd+RVO+TOP	5.22%	-7.99%
mddd+RVO+SAC1	28.87%	24.91%
mddd+mRVO	28.22%	25.27%
mddd+RSVO	18.45%	12.69%
mddd+RSVO+ TOP	-0.42%	-16.63%
mddd+RSVO+ SAC1	26.38%	20.12%
mddd+mRSVO	26.94%	21.68%
wdeg averaged 2,395,567 checks/problem		
wdeg+RVO	4.70%	-1.39%
wdeg+RVO+TOP	-20.29%	-32.10%
wdeg+RVO+SAC1	24.62%	22.83%
wdeg+mRVO	12.41%	16.04%
wdeg+RSVO	-7.78%	-17.35%
wdeg+RSVO+TOP	-25.73%	-35.20%
wdeg+RSVO+SAC1	25.48%	20.70%
wdeg+mRSVO	17.17%	18.00%
dom/wdeg averaged 1,587,777 checks/problem		
dom/wdeg+RVO	9.36%	6.17%
dom/wdeg+RVO+TOP	0.36%	-11.30%
dom/wdeg+RVO+SAC1	26.81%	25.14%
dom/wdeg+mRVO	27.46%	25.49%
dom/wdeg+RSVO	10.42%	3.43%
dom/wdeg+RSVO+TOP	-0.02%	-3.77%
dom/wdeg+RSVO+SAC1	27.33%	22.51%
dom/wdeg+mRSVO	27.12%	22.97%

likelihood of value mistakes near the top of the search tree. First, only for the first focus variable, we use look-ahead (Frost 1995) to choose a value that removes the fewest values from future variables under MAC. Essentially, this is singleton arc consistency (Debruyne and Bessière 1997) restricted to the root of the search tree. We denote it by *SAC1*. Second, we use the same procedure to break ties below the root but within the top of the search tree (its first $ln(n)$ levels, where n is the number of variables in the problem). We denote this modification by *TOP*. All other ties are broken lexically.

We use *mRVO* (or *mRSVO*) to denote *RVO* (or *RSVO*) with both modifications. Observe that, in Figure 2, the curve for *mRSVO* is considerably more uniform than that for *RSVO* on *dom/wdeg* for problems in <50, 10, 0.38, 0.2>. We also tested the survivors-first algorithms on <50, 10, 0.184, 0.631>, a somewhat easier class of the same size. Tables 1 and 4 show that both *mRVO* and *mRSVO* improve all four variable-ordering heuristics on both sets of unstructured problems.

Discussion

To explore the modifications more closely, we ablated the modified versions, testing with each modification in turn. (Results appear in Tables 1 and 4.) On both classes of random problems, *SAC1* is apparently well worth its computation cost; it significantly improves the performance of both *RVO* and *RSVO*. As intended, *SAC1* remedies the lack of information our value heuristics have at the root. Without *RVO* and *RSVO*, however, both maintained singleton arc consistency and *SAC1* alone timed out on these problems. In contrast, *TOP* as the sole modification frequently impaired performance, probably because *RVO* and *RSVO* lacked the experience necessary to avoid value mistakes at the beginning of search. Nonetheless, in Table 1, where the problems are more difficult, *TOP* improves the performance of both *mRVO* and *mRSVO* with *SAC1*. On structured problems (quasigroups and composed problems), however, *SAC1* and *TOP* actually weaken the performance of *RVO* and *RSVO*. (Data omitted.)

Even the low overhead of *RVO* and *RSVO* could unnecessarily degrade performance on easy problems. Therefore, we also tested on a set of 300 smaller but still at the phase transition problems from <30, 8, 0.266, 0.66>. These are solved by *mdd*, *mddd*, *wdeg*, and *dom/wdeg* in fewer than 100,000 checks per problem. (Data omitted.) Both *RVO* and *RSVO* reduced checks for every solver, but by no more than 5%. There were only two small slow downs: *mddd+mRVO* and *mddd+mRSVO* were slower than *mddd* by 6.83% and 8.80%, respectively.

We have begun to compare our results with other value-ordering heuristics as well (Geelen 1992; Ginsberg et al. 1990; Frost and Dechter 1995; Refalo 2004). *Real full look-ahead* as a value-ordering heuristic assigns in turn each value of the focus variable only, and chooses a value

Table 5: Comparison of survivors-first with promise and real full look-ahead. Improvements are positive (reductions); poorer performances (increases) are negative.

	Change in basic performance	
Solver	**Checks**	**CPU time**
Quasigroup 10 × 10 with 67 holes		
dom/wdeg averaged 121,527 checks/problem		
dom/wdeg+promise	–5.00%	–26.00%
dom/wdeg+real full	–115.00%	–43.00%
dom/wdeg+RVO	78.32%	65.83%
dom/wdeg+RSVO	58.10%	47.34%
Quasigroup 10 × 10 with 74 holes		
dom/wdeg averaged 821,802 checks/problem		
dom/wdeg+promise	72.00%	64.60%
dom/wdeg+real full	91.24%	93.86%
dom/wdeg+ RVO	89.99%	88.67%
dom/wdeg+RSVO	96.26%	94.33%
<75, 10, 0.216, 0.05, 1, 8, 10, 0.786, 0.55, 0.104, 0.05>		
dom/wdeg averaged 115,875 checks/problem		
dom/wdeg+promise	47.88%	–232.63%
dom/wdeg+real full	–94.29%	–104.61%
dom/wdeg+RVO	8.02%	3.15%
dom/wdeg+RSVO	11.14%	3.94%
<75, 10, 0.216, 0.05, 1, 8, 10, 0.786, 0.55, 0.012, 0.05>		
dom/wdeg averaged 318,057 checks/problem		
dom/wdeg+promise	9.32%	–154.61%
dom/wdeg+real full	–127.78%	–29.40%
dom/wdeg+RVO	16.00%	6.37%
dom/wdeg+RSVO	9.65%	0.32%
<50, 10, 0.184, 0.631>		
dom/wdeg averaged 1,587,777 checks/problem		
dom/wdeg+promise	38.14%	–28.41%
dom/wdeg+real full	–36.00%	–29.74%
dom/wdeg+mRVO	27.46%	25.49%
dom/wdeg+mRSVO	27.12%	22.97%
<50, 10, 0.38, 0.2>		
dom/wdeg averaged 26,494,236 checks/problem		
dom/wdeg+promise	75.70%	46.03%
dom/wdeg+real full	51.03%	56.67%
dom/wdeg+mRVO	61.69%	58.52%
dom/wdeg+mRSVO	63.22%	62.01%

that removes the fewest values from the domains of all the future variables after AC propagation. *Promise* does look-ahead on the focus variable with forward checking and then chooses a value that maximizes the product of the resultant domain sizes after propagation. Table 5 uses *dom/wdeg* to compare the best of our survivors-first algorithms with promise and real full look-ahead. On four of the six classes cases, real full look-ahead and promise are substantially slower than *dom/wdeg*. On the other two (the harder random and the easier quasigroup), they improve *dom/wdeg* but remain slower than survivors-first. We also intend to apply survivors-first with randomized restart (Harvey and Ginsberg 1995), which was particularly effective with impact (Refalo 2004).

One might expect that unsolvable problems or search for all solutions to a problem would not benefit from value-ordering heuristics (Frost and Dechter 1995). More recent work, however, has shown that 2-way branching and conflict-directed variable-ordering heuristics respond well to value ordering (Smith and Sturdy 2005, Mehta and von Dongen 2005). Under *k-way* branching, another value for the same focus variable is tried after retraction. Under *2-way* branching, consistency is maintained after backtracking, with the possibility of further domain reduction, and a (possibly different) focus variable is selected. We therefore tested our learned value-ordering heuristics under 2-way branching as well. Our learned heuristics continue to perform well under 2-way branching, although there is often less room for improvement. For example, on 100 problems in <50,10,0.38,0.2> with 2-way branching *mddd+mRSVO* reduced *mddd*'s constraint checks by 26.93% and reduced computation time by 39.07% on average.

Setting the tie range very high (near 100%) approaches full look-ahead as the designation for "top of the tree" approaches *n*. Redefining the top of search tree, say as 0.3*n*, has thus far been unsuccessful. We also tried an initial training period with a threshold, measured in number of backtracks, number of visited nodes or number of assigned variables. Before the threshold was reached, values were chosen lexically, without reference to the *R* and *S* values that might be inaccurate early on. After the threshold was reached, we used our value-ordering heuristics; this approach has thus far proved less satisfactory than using *ln(n)*.

We continue to examine the relationship between *SAC1* and our other heuristics. Although our intuition is that survivors-first works best on harder problems, we also intend to evaluate its performance under a full suite of density and tightness values. We are also testing further the decision to exclude values in the domains of the neighbors of the focus variable from the survival computation. Our intuition was that there would be a peculiar bias without the exclusion, because neighbors of the focus variable would be considered only against a single value (the one being assigned).

Implementation of survivors-first in value-oriented AC algorithms like AC-6 may offer further challenges where overhead is linear with the size of the domain (Bessière and Cordier 1993). One possible solution would be to organize weights by search level.

Thus far we have only addressed binary CSPs. We expect, however, that the ideas presented here will smoothly transfer to global constraints. Our empirical results suggest that survivors-first plays a more important role on structured problems, which are more similar to those in the real world.

In summary, our algorithms, *RVO* and *RSVO,* require *O(nm)* space and *O*(*m*) time each time a value is selected, where *n* is the number of variables and *m* is the maximum domain size. Because they make straightforward use of traditional AC algorithms, they incur very little computational overhead. Survivors-first effectively accelerates search directed by four popular variable-ordering heuristics. Both *RVO* and *RSVO* are robust and often provide dramatic speed-up, particularly on the most difficult problems in a set. Their modified versions often further assist in the identification of values that survive to participate in a solution.

References

Bayardo Jr., R. J., and Schrag, R. 1996. Using CSP Look-Back Techniques to Solve Exceptionally Hard SAT Instances. In *Proceedings of CP-1996*, 46-60.

Beck, J. C. 2007. Solution-Guided Multi-Point Constructive Search for Job Shop Scheduling. *JAIR* 29: 49-77

Bessière, C. and Cordier M. O. 1993. Arc-Consistency and Arc-Consistency Again. *Artificial Intelligence* 65: 179-190.

Bessière, C.; Freuder, E. C.; and Régin, J. C. 1999. Using Constraint Metaknowledge to Reduce Arc Consistency Computation. *Artificial Intelligence* 107: 125-148.

Bessière, C., and Régin, J. C. 1996. MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems. In *Proceedings of CP-1996*, 61-75.

Bessière, C., and Régin, J. C. 2001. Refining the Basic Constraint Propagation Algorithm. In *Proceedings of IJCAI-2001*, 309-315.

Bessière, C.; Régin, J. C.; Yap, R. H. C.; and Zhang, Y. 2005. An Optimal Coarse-Grained Arc Consistency Algorithm. *Artificial Intelligence* 165: 165-185.

Boussemart, F.; Hemery, F.; Lecoutre, C.; and Sais, L. 2004. Boosting Systematic Search by Weighting Constraints. In *Proceedings of ECAI-2004*, 146–150.

Debruyne, R., and Bessière, C. 1997. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In *Proceedings of IJCAI-1997*, 412–417.

Dechter, R., and Pearl, J. 1988. Network-based Heuristics for Constraint Satisfaction Problems. *Artificial Intelligence* 34: 1-38.

Epstein, S. L.; Freuder, E. C.; and Wallace, R. L. 2005. Learning to Support Constraint Programmers. *Computational Intelligence* 21: 337 – 371.

Frost, D., and Dechter, R. 1995. Look-ahead Value Ordering for Constraint Satisfaction Problems. In *Proceedings of IJCAI-1995*, 572–578.

Geelen, P. A. 1992. Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems. In *Proceedings of ECAI-1992*, 31-35.

Gent, I. P.; MacIntyre, E.; Prosser, P.; and Walsh, T. 1996. The Constrainedness of Search. In *Proceedings of AAAI-1996*, 246-252.

Ginsberg, M.; Frank, M.; Halpin, M.; and Torrance, M. 1990. Search Lessons Learned from Crossword Puzzles. In *Proceedings of AAAI-1990*, 210–215.

Golomb, S., and Baumert, L. 1965. Backtracking Programming *Journal of the ACM* 12: 516-52.

Gomes, C.; Fernandez, C.; Selman, B.; and Bessière, C. 2004. Statistical Regimes Across Constrainedness Regions. In *Proceedings of CP-2004*, 32–46, Springer.

Harvey, W. D. 1995. Nonsystematic Backtracking Search. PhD Thesis, Stanford University (1995).

Harvey, W. D., and Ginsberg, M. L. 1995. Limited Discrepancy Search. In *Proceedings of IJCAI-1995*, 607-613.

Hulubei, T., and O'Sullivan, B. 2005. Search Heuristics and Heavy-Tailed Behavior. In *Proceedings of CP-2005*, 328-342.

Kask, K.; Dechter, R.; and Gogate, V. 2004. Counting-Based Look-Ahead Schemes for Constraint Satisfaction. In *Proceedings of CP-2004*, 317-331.

Lecoutre, C.; Sais, L.; Tabary, S.; and Vidal, V. 2007. Nogood Recording from Restarts. In *Proceedings of IJCAI-2007*, 131-136.

Mackworth, A. K. 1977. Consistency in Networks of Relations. *Artificial Intelligence* 8: 99–118.

Mehta, D., and von Dongen, M.R.C. 2005. Static Value Ordering Heuristics for Constraint Satisfaction Problems. In *Proceedings of CPAI-2005*, 49–62.

Meisels, A.; Shimony, S. E.; and Solotorevsky, G. 1997. Bayes Networks for Estimating the Number of Solutions to a CSP. In *Proceedings of AAAI-1997*, 185-190.

Meseguer, P., and Walsh, T. 1998. Interleaved and Discrepancy Based Search. In *Proceedings of ECAI-1998*, 239-243.

Minton, S. 1996. Automatically Configuring Constraint Satisfaction Programs: A Case Study. *Constraints* 1: 7-44.

Refalo, P. 2004. Impacted-Based Search Strategies for Constraint Programming. *Proceedings of CP-2004*, 557-571.

Walsh, T. 1997. Depth-bounded Discrepancy Search, In *Proceedings of IJCAI-1997*, 1388-1393.

Sabin, D., and Freuder, E. C. 1994. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of ECAI-1994*, 125-129.

Smith, B. M., and Sturdy, P. 2005. Value Ordering for Finding All Solutions. In *Proceedings of IJCAI-2005*, 311–316.

Vernooy, M., and Harvens, W. S. 1999. An Evaluation of Probabilistic Value-Ordering Heuristics. in *Proceedings of the Australian Conference on Artificial Intelligence*, 340-352.

Zanarini, A., and Pesant, G. 2007. Solution Counting Algorithms for Constraint-Centered Search Heuristics. In *Proceedings of CP-2007*, 743-757.