Preferences Improve Learning to Solve Constraint Problems

Smiljana Petrovic[1] and Susan L. Epstein[1,2]

[1]Department of Computer Science, The Graduate Center of The City University of New York, NY, USA
[2]Department of Computer Science, Hunter College of The City University of New York, NY, USA
spetrovic@gc.cuny.edu, susan.epstein@hunter.cuny.edu

Abstract

Search heuristics that embody domain knowledge often reflect preferences among choices. This paper proposes a variety of ways to identify a good mixture of search heuristics and to integrate the preferences they express. Two preference expression methods inspired by the voting literature in political science prove particularly reliable. On hard problems, good methods to integrate such heuristics are shown to speed search, reducing significantly both computation time and the size of the search tree.

Introduction

For complex search problems, several heuristics are often available. Although using more than one heuristic often proves more efficient (Valentini and Masulli, 2002), precisely how to combine heuristics is not well understood. The thesis of this work is that, when each heuristic reflects an underlying metric, nuances from comparative opinions (here, *heuristics' preferences*) can be exploited to improve search performance. This paper is about two kinds of preferences: a solver's preferences for heuristics, and the heuristics' preferences for actions during search.

We explore here a variety of ways to express and exploit heuristics' preferences. These approaches consider both the scores returned by the metrics on which these heuristics rely and the distributions of those scores across a set of possible actions. Our principle results are that both preferences for heuristics and preferences expressed by heuristics can significantly improve heuristic-guided search.

Our domain of investigation is constraint satisfaction problems (*CSP*s). A CSP is a set of variables, each with a domain of values, and a set of constraints, expressed as relations over subsets of those variables. A *solution* to a CSP is an instantiation of all its variables that satisfies all the constraints. The knowledgeable reader is forewarned that, despite the CSP context, this paper does not use "preferences" as traditionally employed, in soft constraints. There, preferences on constraints are used to optimize a solution; here, preferences guide search in two different ways: preferences *for* heuristics allocate resources, and preferences *of* heuristics make choices. Each of our heuristics assigns a number to each possible search decision; that number is the *preference* of the heuristic for the choice.

The test-bed here is *ACE* (the Adaptive Constraint Engine). It learns a customized combination of pre-specified search-ordering heuristics for a class of CSPs (Epstein, Freuder, et al., 2005). ACE's heuristics are gleaned from the CSP literature. Each heuristic has its own underlying descriptive *metric*, a function from the set of available choices to the real numbers. That metric returns a *score* for each choice. The heuristic then expresses its preferences for actions whose scores are most in agreement with its own predilection. (For example, *min-domain* uses a metric that calculates the domain size of each variable; the heuristic then prefers those variables with the smallest domain size.) In this way each of ACE's heuristics indicates its degree of preference for each available action. To make a decision during search, ACE uses a weighted mixture of expressions of preference from a large number of such heuristics. During learning, ACE refines those weights after each problem.

Scores returned by the heuristics' metrics, however, can be very large or very small. Prior to the work described here, ACE's heuristics simply expressed their preferences as a rank on a common scale. We propose here five alternative methods to express preferences, and test them on a variety of problems. In addition, prior to this work ACE included in the mixture learned for a problem class all heuristics with better than random performance. We also investigate here further reduction in the number of heuristics included in the mixture.

The next two sections provide background on constraint satisfaction and related work on combining heuristics. The paper then describes the architecture used to combine heuristics. In subsequent sections we propose methods that exploit heuristics' preferences, describe recent experiments, and discuss the results.

Constraint satisfaction problems

A *problem class* is a set of CSPs with the same characterization. For example, CSPs in *model B* are characterized by <*n*,□,□,□>, where *n* is the number of variables, *m* the maximum domain size, *d* the density (fraction of edges out of *n(n-1)/2* possible edges) and *t* the *tightness* (fraction of possible value pairs that each constraint excludes) (Gomes, Fernandez, et al., 2004). In a *binary* CSP, all constraints are on at most two variables. A binary CSP can be represented as a *constraint graph*, where vertices correspond to the variables (labeled by their domains), and each edge represents a constraint between its respective variables. (Because their number of variables and domain sizes uniquely identifies the model B classes used here, we refer to them simply as *n-m*.)

A problem class can also mandate some non-random structure on its problems. For example, a *composed prob-*

lem consists of a subgraph called its *central component* loosely joined to one or more subgraphs called *satellites* (Aardal, Hoesel, et al., 2003). The class of composed problems used here is referred to as *Comp*.

Traditional CSP search iteratively selects a variable and assigns it a value from its domain. After each assignment, some form of *inference* detects values that are incompatible with the current instantiation. All experiments reported here used the MAC-3 inference algorithm to maintain arc consistency during search (Sabin and Freuder, 1997). MAC-3 temporarily removes currently unsupportable values to calculate *dynamic domains* that reflect the current instantiation. If every value in any variable's domain is *inconsistent* (violates some constraint), the current instantiation cannot be extended to a solution, so some *retraction* method is applied. Retraction in all experiments reported here *backtracks chronologically*: the subtree rooted at the inconsistent node (*digression*) is pruned, and the most recent value assignment(s) withdrawn.

Search for a CSP solution is an NP-complete problem; the worst-case cost is exponential in *n* for any known algorithm. Often, however, a CSP can be solved with a cost much smaller than the worst case. Although CSPs in the same class are ostensibly similar, there is evidence that their difficultly may vary substantially for a given search algorithm (Hulubei and O'Sullivan, 2005). All data reported here was, of necessity, generated under a generous *step limit* (maximum number of search decisions) to control resources and measure search tree size. All cited differences in this paper are statistically significant at the 95% confidence level.

CSP heuristics direct the selection of the next variable and its value. Typically such a heuristic reports only the choice with the highest preference and requires a tie-breaking mechanism. Of necessity, this paper references many CSP heuristics. A full listing of their underlying metrics and associated definitions appears in the appendix.

Combining heuristics

There are several reasons why a combination of experts can offer improved performance, compared to a single expert (Valentini and Masulli, 2002). If no single expert is best on all the problems, a combination of experts could enhance the accuracy and reliability of the overall system. On limited training data, different candidate heuristics may appear equally accurate. In this case, one could better approximate the unknown, correct heuristic by averaging or mixing candidate heuristics, rather than selecting one (Dietterich, 2000). For supervised learning algorithms, the performance of such a *mixture of experts* has been theoretically analyzed in comparison to the best individual expert (Kivinen and Warmuth, 1999). Under the worst-case assumption, even when the best expert is unknown (in an online setting), mixture-of-experts algorithms have been proved asymptotically close to the behavior of the best expert (Kivinen and Warmuth, 1999).

Selection of appropriate heuristics from the many touted in the constraint literature is non-trivial. Table 1 illustrates that even well-trusted individual heuristics vary dramatically on their performance on different classes. For example, *max-weighted-degree* (Boussemart, Hemery, et al., 2004) is among the best individual heuristics on model B random problems where the number of variables is substantially larger than the maximum domain size (e.g., 50-10). It appears to be less effective, however, when there are more potential values than variables (e.g., 20-30).

Table 1: Search tree size under individual heuristics on 50 problems from each of four classes. Heuristic definitions appear in the Appendix.

Heuristic	***50-10***	***20-30***	***30-8***
min-domain	51,347	10,411	563
max-degree	46,347	5,267	206
max-forward-degree	43,890	10,150	220
min-domain/degree	35,175	4,194	234
max-weighted-degree	30,956	5,897	223
min-dom/dynamic-deg	30,791	3,942	211
min-dom/weighted-deg	30,025	4,090	205

A *dual* for a heuristic reverses the import of its metric (e.g., *max-domain* is the dual of *min-domain*). Duals of popular heuristics can be superior to traditional heuristic on real-world problems and problems with non-random structure (Lecoutre, Boussemart, et al., 2004; Otten, Grönkvist, et al., 2006; Petrie and Smith, 2003). For example, a composed problem whose central component is substantially larger, looser (has lower tightness) and sparser (has lower density) than its satellite is particularly difficult for some traditional heuristics. Composed problems are typically solved either with minimal backtracking or go unsolved after many thousands of steps. For example, the traditional *max-degree* heuristic (prefer variables with the largest degree in the constraint graph) tends to select variables from the central component for some time, until, deep within the search tree, inconsistencies eventually arise with satellite variables. In contrast, the decidedly untraditional *min-degree* heuristic tends to prefer variables from

Table 2: Performance of 3 popular heuristics (in italics) and their duals on 50 composed problems (described in the text) under a 100,000 step limit. Observe how much better the duals perform on problems from this class.

Heuristic	***Unsolved problems***	***Steps***
Max degree	*9*	*19901.76*
Min degree	0	64.60
Max forward-degree	*4*	*10590.64*
Min forward-degree	0	64.50
Min domain/degree	*7*	*15558.28*
Max domain/degree	4	10922.82

Table 3: Search tree size under individual heuristics and under mixtures of heuristics on three classes of problems. Note that each class has its own particular combination of more than two heuristics that performs better.

Mixture	*50-10*	*20-30*	*30-8*
The best representation from Table 1	30,025	3,942	205
Min dom/dynamic degree + Max Product Domain Value	15,091	2,764	156
Max-weighted-degree + Max Product Domain Value	22,273	3,892	179
Mixture found by ACE	12,120	2,502	141

the satellite and thereby detects inconsistencies much earlier. Table 2 shows how traditional heuristics and their duals fare on composed problems. Surprisingly, the simplest duals do by far the best. We emphasize that the characteristics of such composed problems are often found in real-world problems as well. To achieve good performance without knowledge about a problem's structure, therefore, it is advisable to consider many popular heuristics along with their duals.

A good mixture of heuristics can outperform even the best individual heuristic, as Table 3 demonstrates. The first line shows the best performance achieved by any traditional single heuristic from Table 1. The second line of Table 3 show that a good pair of heuristics, one for variable ordering and the other for value ordering, can perform significantly better than an individual heuristic. Nonetheless, the identification of such a pair is not trivial. For example, *max-product-domain-value* better complements *min-domain/dynamic-degree* than it does *max-weighted-degree*. The last line of Table 1 demonstrates that combinations of more than two heuristics can further improve performance. This paper furthers work on the automatic identification of particularly effective mixtures.

The test-bed

The test-bed used here, ACE, is based on FORR, an architecture for the development of expertise from multiple heuristics (Epstein, 1994). ACE learns to solve a class of CSPs. It customizes a weighted mixture of pre-specified heuristics for any given class (Epstein, et al., 2005). Each heuristic is implemented as a procedure called an *Advisor*. Advisors either address the next variable to be assigned a value (*variable-ordering heuristics*) or the next value to assign to that variable (*value-ordering heuristics*).

Guided by its Advisors, ACE solves problems in a given class and uses its search experience to learn a *weight profile* (a set of weights for the Advisors). To make a decision (select a variable or a value), ACE consults all its Advisors.

Each Advisor A_i expresses the *strength* s_{ij} of its preference for choice c_j. Based on the weight profile, the choice with the highest weighted sum of Advisors strengths to choices is selected:

$$\arg\max_j \sum_i w_i s_{ij} \qquad [1]$$

Initially, all weights are set to 0.05. During learning, however, ACE gleans training instances from its own (likely imperfect) successful searches. *Positive training instances* are those made along an error-free path extracted from a solution. *Negative training instances* are value selections at the root of an incorrect assignment (a *digression*), as well as variable selections after which a value assignment fails. Decisions made within a digression are not considered. ACE's weight learning algorithm, *RSWL* (Relative Support Weight Learning), uses those training instances to update the weight profile. Thus it refines its search algorithm before it continues on to the next problem.

RSWL considers all heuristics' preferences when a decision is made (Petrovic and Epstein, 2006b). Weight reinforcements under RSWL depend upon the normalized difference between the strength the Advisor assigned to that decision and the average strength it assigned to all available choices. This results in a reward when the difference is positive, and a penalty when the difference is negative.

The CSPs we address here are non-trivial. We therefore bring to bear on them two powerful options available with ACE: full restart and random subsets.

- Occasionally, on a challenging problem class, class-inappropriate heuristics acquire high weights on a problem early in training, and then control subsequent decisions. If this results in repeated failure to solve problems, ACE resorts to *full restart*: it recognizes that its current learning attempt is not promising, abandons the responsible training problems, and restarts the entire learning process with a freshly-initialized weight profile (Petrovic and Epstein, 2006a).
- Given an initial set of heuristics that is large and inconsistent, many class-inappropriate heuristics may combine to make bad choices, and thereby make it difficult to solve the problem within a given step limit. Because only solved problems provide training instances for weight learning, no learning can take place until some problem is solved. *Random subsets* have proved a successful approach to this issue; rather than consult all its Advisors at once, ACE randomly selects a new subset of Advisors for each problem, consults them, makes decisions based on their *comments* (expressions of preference), and updates only their weights (Petrovic and Epstein, 2007).

How to exploit heuristics' preferences

Each Advisor takes a heuristic view of the current search choices based on its own descriptive metric. A metric can return large scores, for example, the size of a potential search tree after some value assignment, estimated as the product of all the dynamic domain sizes. A metric can also return small scores, for example, the likelihood of search failure estimated as the average tightness over the neighbors in the original constraint graph to the power of the original degree of the variable. To combine heuristics, therefore, scores returned by metrics should be scaled into some common range. We illustrate here several ways of doing so.

Advisor *A* calculates some score on each choice *c* in the set *C* of all current choices. In this way, an Advisor's metric partitions *C* into *t* subsets C_1, C_2,…, C_t where choices in the same subset share a common score. These subsets are ordered decreasingly by their Advisor's predilection: if *A* prefers choices from C_i to choices from C_j, then $i < j$.

In all the methods considered here for expressing heuristics' preferences, the number of choices on which an Advisor comments depends upon the parameter *p*. An Advisor assigns positive strengths to choices from the first *p* subsets. Choices from C_k for $k > p$ are given strength 0; this makes them irrelevant during voting. For example in Table 4, if a heuristic is presented with 30 choices and $p = 5$, it is expected to comment on those choices with the 5 highest scores, 18 choices in all. Equation [1] expects that those comments will be accompanied by some strength, however. The *strength s* (s_{ij} in equation [1]) of *A*'s support for choice *c* is calculated by applying some function *f* to the score v_k associated with the subset C_k that contains *c*:

$$\forall c \in C_k, s(c) = f(v_k) \qquad [2]$$

We address the impact of *f* with six different approaches. A sample use of each preference expression method on the same set of scores appears in Table 4.

***Ranking*:** Use v_k to order the choices. Assign strength *p* to all choices from C_1, *p*–1 to the choices from C_2, and so on:

$$(\forall c \in C_k, k \le p)\ \textbf{\textit{rank}}(v_k) = p - k + 1 \qquad [3]$$

Ranking is ACE's original preference expression method. It reflects the preferences of one choice over another, but it loses information contained in the score v_k in two ways. First, it ignores the extent to which one choice is preferred over another. For example in Table 4, if the metric were for variables' domain sizes, c_1 and c_3 differ by 25, while the domain sizes of c_3 and c_{13} differ by only one. Nonetheless, ranking assigns equally distant strengths (5, 4 and 3) to those variables. Second, ranking ignores how many choices share the same score. For example in Table 4, the ranks of choices c_3 and c_{13} differ by only 1, although the heuristic prefers only 2 choices over c_3 and 12 choices over c_{13}.

Linear interpolation: Linear interpolation not only considers the relative position of choices, but also the actual differences between scores. Strength differences are made proportional to the differences in scores. The strength for choice *c* is determined by a linear function on its score v_k through points (v_1, *p*) and (v_p, 1).

$$(\forall c \in C_k, k \le p)\ \textbf{\textit{linear}}(v_k) = \frac{p-1}{v_1 - v_p}(v_k - v_p) + 1 \qquad [4]$$

Here, the strengths for choices are real numbers in the interval [1, p]. Choices in C_1 have strength *p*, choices in C_p have strength 1, and the strengths of other choices are linearly interpolated from the scores of their underlying metric. For example, in Table 4, the strengths are determined by the value of the linear function through points (50, 5) and (10, 1).

Exponential scaling based on rank: Exponential scaling emphasizes choices with large metric scores. In the first variation, the rank of each choice is scaled exponentially:

$$(\forall c \in C_k, k \le p)\ \textbf{\textit{exp-rank}}(v_k) = a \cdot b^{rank(v_k) - rank(v_1)} \qquad [5]$$

where *a* and *b* are constants. For example, in Table 4, $a = 10$, $b = 2$, and the strength 10 of the highest-scoring choices is halved for each subsequent subset.

Exponential scaling based on linear interpolation: The second variation on exponential scaling emphasizes choices with the highest metric score as well as relative strengths. It scales the linearly interpolated score of each choice exponentially:

$$(\forall c \in C_k, k \le p)\ \textbf{\textit{exp-linear}}(v_k) = a \cdot b^{linear(v_k) - linear(v_1)} \qquad [6]$$

For example, in Table 4, with $a = 10$, $b = 2$, the strength 10 of the highest-scoring choices in C_1 is divided by $2^{2.5}$ to obtain the strength for the choices in C_2, and by $2^{2.6}$ for the choices in C_3.

Borda methods: The Borda election method was devised

Table 4: An example of how different preference expression methods impact a single metric. Strengths that express preferences over 30 available choices are calculated under 6 different methods. $|C_i|$ is the total number of choices with the score v_i. Only the 5 highest scores were under consideration here.

Choices	c_1-c_2	c_3-c_{12}	c_{13}	c_{14}-c_{16}	c_{17}-c_{18}	c_{19}-c_{20}	c_{21}-c_{30}
Score v_i	50	25	24	15	10	3	2
Subset size $\|C_i\|$	2	10	1	3	2	2	10
Method	**Strengths**						
rank	5	4	3	2	1	0	0
linear	5.00	2.50	2.40	1.50	1.00	0.00	0.00
exp–rank	10.00	5.00	2.50	1.25	0.63	0.00	0.00
exp–linear	10.00	1.77	1.65	0.88	0.63	0.00	0.00
Borda–up	30.00	28.00	18.00	17.00	14.00	0.00	0.00
Borda–down	28.00	18.00	17.00	14.00	12.00	0.00	0.00

by Jean-Charles de Borda in 1770 (Brams and Fishburn, 2002). It inspired the next two preference expression methods. Borda methods consider the total number of available choices and the number of choices with a smaller score. Thus the strength for a choice is based on its position relative to the other choices. Our first variation on Borda sets strength equal to the number of choices scored no higher than this choice:

$$(\forall c \in C_k, k \leq p)\ \textbf{\textit{Borda – up}}\ (v_k) = \sum_{j=k}^{t} |C_j| \qquad [7]$$

where t is the number of subsets created by the partition. For example, in Table 4, the 2 highest-scoring choices (in C_1) are assigned strength 30, the next highest (in C_2) are assigned strength 30–2=28, and so on.

Our second variation on Borda assigns one point for each choice with a lower score.

$$(\forall c \in C_k, k \leq p)\ \textbf{\textit{Borda – down}}\ (v_k) = \sum_{j=k+1}^{t} |C_j| \qquad [8]$$

The difference between the two Borda methods is evident when many choices share the same score. Borda–up considers how many choices score higher. A large subset results in a big gap in strength between that subset and the next (less preferred) one. Borda–down considers only how many choices score lower, so that a large subset results in a big gap in strength between that subset and the previous (more preferred) one. In Table 4, for example, 10 choices share the second-highest score. In Borda–up, that results in a 10-point difference between the strengths of C_2 and C_3 (28 vs. 18). In Borda–down, there is a 10-point difference between the strengths of C_1 and C_2 (again 28 vs. 18).

Experimental Design

We tested these six approaches to heuristics' preferences with ACE on three classes of randomly-generated, model B CSPs: <50,10,0.38,0.2> with many variables and relatively small domains; <20,30,0.444,0.5> with fewer variables but larger domains, and the somewhat easier <30,8,0.26,0.34>. (In the tables they are referred to as *50-10*, *20-30*, and *30-8*, respectively.) All these problem classes are at the phase transition, that is, particularly difficult for their size (n and m). We also tested on *Comp,* a class of composed problems, where the central component was model B with <22, 6, 0.6, 0.1>, linked to a single model B satellite with <8, 6, 0.72, 0.45> by edges with density 0.115 and tightness 0.05. These are challenging but solvable problems; some of them appeared in the First International Constraint Solver Competition at CP-2005.

For each experiment, we report the search tree size during testing, averaged over 10 runs. A *run* here is a learning phase, during which ACE adjusts its weight profile and attempts to solve problems with its Advisors, followed by testing phase, during which the weights are frozen and some weight-based subset of the Advisors is consulted. The weight-learning algorithm was provided with 42 Advisors, described in the Appendix: 28 for variable ordering and 14 for value ordering. We used random subsets to speed learning: for each problem in the learning phase, a random number r in [0.3,0.7] was generated, and then r of the variable-ordering Advisors and r of the value-ordering Advisors, were selected without replacement to make decisions during search on that problem. We also had ACE monitor learning reliability with full restart: during the learning phase, failure on 8 of the last 10 tasks triggered a full restart. At most 2 full restarts (3 attempts) were permitted. During the learning phase, ACE was required to try to solve 30 problems from the first solved problem in its current full-restart attempt. Problems were never reused during learning, even under full restart. During learning, step limits on individual problems were set high enough to allow ACE to find solutions, but not so high that search wandered and thereby produced poor training examples and learned poor weight profiles. Here, the step limits on individual problems were 100,000 for 50-10, 10,000 for 20-30, 2,000 for 30-8 and 1,000 for *Comp* problems.

During ACE's *testing phase* it attempts to solve a sequence of fresh problems with learning turned off. For each problem class, every testing phase used the same 50 problems. (Those problems were also used in Tables 1 through 3.) The step limit on each testing problem was 100,000 steps, except for the composed problems, where it was 10,000.

To examine preferences *for* heuristics, the first group of experiments all used ranking (equation [2]) but reduced the number of Advisors in testing, and made decisions based only on a pre-specified number of the top-weighted Advisors. This is in contrast to ACE's original approach, which included during testing every Advisor whose weight is higher than that of its respective benchmark. (A *benchmark* expresses random preferences over the same set of choices an Advisor faces. Benchmarks are excluded from decision-making.)

To examine the preferences *of* heuristics, the second group of experiments tested the preference expression methods described in the previous section, as defined in equations [2] through [8]. Advisors commented on choices with the p=5 highest scores. In exponential scaling methods, the scaling constant was a=10 and the base of the exponential function was b=2. During testing, a pre-specified number of Advisors with the highest weights was consulted.

Results and discussion

Preferences *for* heuristics

The first set of experiments sought to speed search by the elimination of more heuristics during testing. The overall time to solve a problem depends upon the number of Advisors consulted, the computational cost of each of those Advisors, and the number of decisions on which an Advisor is consulted. Ideally, one would reduce the number of Advisors and avoid costly (i.e., resource-greedy) Advisors with-

out increasing the size of the search tree. Often, however, costly Advisors are worthwhile, since they produce valuable information.

In ACE's traditional approach, the benchmark criterion typically eliminates about half the initial Advisors. (Here, 16 out of 28 variable-ordering Advisors and 6 out of 14 value-ordering Advisors usually survived the benchmark criterion on any given run.) These experiments began with the 8 highest-weighted variable-ordering and the 4 highest-weighted value-ordering Advisors that met the benchmark criterion, thereby further halving the number consulted during testing.

Table 5 reports the average search tree size during testing with fewer Advisors. If any problems went unsolved within the step limit, those are reported in Table 5 as well. An initial reduction, to 8 variable-ordering and 4 value-ordering Advisors, did not have a statistically significant impact. With more extensive reductions, however, the search tree sizes for the 20-30 and 30-8 problems eventually increased. For the 50-10 problems, the search tree size remained stable. We believe the explanation lies in the nature of the problems themselves. When there are many values compared to the number of variables, despite inference with MAC-3, domains remain large and many values still share the same scores (and strengths). With so few value-ordering Advisors, ties among value choices occur more often, so that random selection among tied values is more likely.

The *Comp* problems were negatively affected by the use of fewer Advisors. Recall that although these are solvable problems, they are typically solved either with minimal search or go unsolved after extensive search. Even inappropriate heuristics solve some of these problems quickly. As result, such heuristics may achieve erroneously-high weights under RSWL. When the number of Advisors is reduced, the influence of each surviving heuristic increases, and overall performance can be worse.

The computation time required by an Advisor depends on its metric and its commonalities with other Advisors. Some Advisors have readily-computed metrics (e.g. *min-domain*), while others' metrics require substantial calculation time (e.g. *min-product-domain-value* uses the products of the domain sizes of the neighbors after each potential value assignment). Moreover, some Advisors' metrics are based upon metrics already calculated for other Advisors (e.g. *dynamic-domain/weighted-degree* is based on *weighted-degree* and *dynamic-domain*). If such Advisors are consulted together, the total computation time can be close to the time for some subset of them.

Table 6 compares the running time with fewer Advisors to the time required by the traditional approach, which uses every Advisor whose weight is greater than the weight of its benchmark Advisor. Greater speedup occurred when more value-ordering Advisors were eliminated, because their metrics tend to be more costly. The exception here is the *Comp* problems, where the increased search tree size prevents any speedup.

Table 5: Search tree size for 4 CSP classes when the number of Advisors during testing is reduced. Each statistically significant reduction in performance compared to the traditional benchmark approach (> bmk) appears in bold.

Var. Adv.	Val. Adv.	***50-10*** *Steps*	*Failed*	***20-30*** *Steps*	***30-8*** *Steps*	***Comp*** *Steps*	*Failed*
>bmk	>bmk	19,689	4.4	2,864	187	450	1.6
8	4	19,923	3.7	2,956	188	572	2.4
8	2	19,372	3.4	2,930	190	**679**	2.8
4	4	19,265	3.2	**3,198**	**207**	**1,064**	4.6
4	2	19,428	3.2	**3,176**	**206**	**934**	4.1

Preferences *of* heuristics

In the second set of experiments, we investigated different ways to express heuristics' preferences. The strengths of Advisors' preferences are crucial in making a decision. Strong preferences from low-weight Advisors can easily overpower a slight preference from a high-weight Advisor. Heuristics' preferences were combined with linear interpolation (which respects the relative preferences of heuristics), or with exponentiation on ranks (which strongly favors each heuristic's top-scoring choice), or with a Borda method (which emphasizes the relative position of a choice among other choices).

During testing, ACE used its eight highest-weighted variable-ordering Advisors and two highest-weighted value-ordering Advisors. As observed above, for model B problems, that combination had produced substantial speedup without any significant change in the search tree size (Tables 5 and 6). Search tree size for these three classes under the six preference expression methods appear in Table 7. Because fewer Advisors eventually degraded performance on the *Comp* problems, we tested preference expression on reduction with the benchmark criteria and on eight variable-ordering Advisors and two value-ordering Advisors. Those results appear in Table 8. Figures in bold represent a statistically significant improvement over to ranking.

The greatest improvements came on difficult problems

Table 6: Percent of computation time compared to the traditional (>bmk) approach when the number of Advisors during testing is reduced.

Var. Adv.	***Val. Adv.***	***50-10***	***20-30***	***30-8***	***Comp***
>bmk	>bmk	100%	100%	100%	100%
8	4	84%	71%	87%	95%
8	2	70%	55%	68%	106%
4	4	77%	87%	100%	158%
4	2	69%	67%	78%	122%

with Borda–up, Borda–down, and linear preference expression. On the 50-10 problems, the weights learned under the Borda methods actually reduced the search tree size to less than half what a traditional, off-the-shelf heuristic had done. This demonstrates the power of exploitation of the nuances of preference information. No such improvement in search tree size was detected on either of the easier random problem classes.

Under Borda–up, if only a few choices score higher, the strength of the choices from the next lower-scoring subset is close enough to influence the decision. If there are many high-scoring choices, the next lower subset will have much lower strength, which decreases its influence. With Borda–down, when many choices share the same score, they are penalized for failure to discriminate, and their strength is lowered. On random problems, there is no significant difference between those two Borda methods. On *Comp* problems, however, that is exactly the difference that improves the search (Table 8). A good combination of heuristics for *Comp* problems forces the subproblem defined by the satellite to be solved first. It appears that there are many ties in the relatively large central component, so that the subsets of choices with the same scores are relatively large as well. When Borda–down assigns lower strengths to large subsets from the central component, it makes them less attractive. Table 8 also shows that reducing the number of Advisors decreases performance on *Comp* problems with every preference expression method, and that the relative performance of these methods on *Comp* problems is unaffected by using fewer Advisors.

From one run to the next, both the linear and the Borda methods were consistent learners, that is, there was little variation in the search tree size or in the Advisors selected. For example, on 50-10 problems with Borda–down, the average search tree size per run ranged from 12,119.46 to 16,546.66, and in 6 out of 10 runs the same 8 Advisors had the highest weight. (The other 4 runs included 7 of them.) With exponential preference expression, there was considerably more variation. For example, on 50-10 problems with exp–rank, the average search tree size per run ranged from 15,252.00 to 25,594.84.

The two exponential methods dramatically reduce the influence of low-scoring choices. As a result, the top choices of the highest-weighted Advisors are more likely to dominate the decision process. Because the RSWL algorithm bases its credits and penalties on the deviation from each Advisor's average strength, exponential methods incur particularly harsh penalties on an Advisor whose metric errs.

Future research will combine preference expression methods with a variety of weight learning algorithms. In addition to Advisors' preferences, RSWL can consider other factors to determine when and how much to reinforce weights (e.g., the constrainedness of a subproblem, the number of available choices, and the depth of the search subtree). Understanding the interaction of those factors with preference expression methods and their parameters (e.g., the number of subsets *p* considered, or the base of the exponential function) could further improve search and learning performance.

Meanwhile, the work reported here illustrates the impact preference expression can have on heuristic mixtures. Making informed decisions speeds search, particularly on difficult problems. Mere ranking ignores the degree of metric difference. Exponential methods, which stress choices with higher scores, reduce the influence of low-scoring choices dramatically. In contrast, the linear, Borda–up and Borda–down methods use the nuances of preference information: both relative difference and relative positions among scores. When preference for fewer Advisors is combined with more sensitive expression of those Advisors' preferences, it is possible to significantly reduce both computation time and the size of the search tree on difficult problems.

Table 7: Search tree size for 3 model B CSP classes under 6 preference expression methods with 8 variable-ordering and 2 value-ordering Advisors. Statistically significant improvements over ranking appear in bold.

	50-10		***20-30***	***30-8***
Method	*Steps*	*Failed*	*Steps*	*Steps*
rank	19,372.27	3.40	2,929.97	189.95
linear	**15,385.27**	2.30	2,859.77	**152.74**
exp_rank	19,916.80	3.10	2,700.16	**170.46**
exp_linear	20,657.03	3.10	2,695.39	188.50
Borda–up	**14,480.07**	1.90	2,941.70	175.08
Borda–down	**14,338.02**	2.30	3,091.35	175.70

Appendix

The metrics underlying ACE's heuristic tier-3 Advisors were drawn from the CSP literature. All are computed dynamically, except where noted. One vertex is the *neighbor*

Table 8: Search tree size for *Comp* class under a variety of selection regimes. Statistically significant improvements over ranking appear in bold.

	Benchmark criterion		***8 variable Advisors 2 value Advisors***	
Method	*Steps*	*Failed*	*Steps*	*Failed*
rank	449.62	1.60□	679.36	2.80
linear	589.77	2.10	959.49	3.80
exp_rank	737.23	3.10	838.84	3.80
exp_linear	413.10	1.60	537.32	2.10
Borda–up	563.74	2.00	672.30	2.40
Borda–down	**264.87**	0.70	**369.04**	1.20

of another if there is an edge between them. A *nearly neighbor* is the neighbor of a neighbor in the constraint graph (but not the variable itself). The *degree of an edge* is the sum of the degrees of the variables incident on it. The *edge degree* of a variable is the sum of edge degrees of the edges on which it is incident. Each metric produces two Advisors.

Metrics for variable ordering:

• Number of neighbors in the constraint graph (static)
• Number of remaining possible values
• Number of unvalued neighbors
• Number of valued neighbors
• Ratio of dynamic domain size to degree
• Number of constraint pairs for variable (Kiziltan, Flener, et al., 2001)
• Edge degree, with preference for the higher/lower degree endpoint
• Edge degree, with preference for the lower/higher degree endpoint
• Dynamic edge degree, with preference for the higher/lower degree endpoint
• Dynamic edge degree, with preference for the lower/higher degree endpoint

Metrics for value ordering:

• Number of variables already assigned this value
• Number of value pairs on the selected variable that include this value
• Minimal resulting domain size among neighbors after this assignment (Frost and Dechter, 1995)
• Number of value pairs from neighbors to nearly neighbors supported by this assignment
• Number of values among nearly neighbors supported by this assignment supported by this assignment
• Domain size of neighbors after this assignment, breaking ties with frequency (Frost and Dechter, 1995)
• Weighted function of the domain size of the neighbors after this assignment, a variant on an idea in (Frost and Dechter, 1995)
• Product of the domain sizes of the neighbors after this assignment

Bibliography

Boussemart, F., F. Hemery, C. Lecoutre and L. Sais (2004). Boosting Systematic Search by Weighting Constraints. *ECAI 2004*, pp. 146-150.

Brams, S. J. and P. C. Fishburn (2002). Voting procedures. *Handbook of Social Choice and Welfare* Volume 1: 173-236.

Epstein, S. L. (1994). For the Right Reasons: The FORR Architecture for Learning in a Skill Domain. *Cognitive Science* 18: 479-511.

Epstein, S. L., E. C. Freuder and R. Wallace (2005). Learning to Support Constraint Programmers. *Computational Intelligence* 21(4): 337-371.

Frost, D. and R. Dechter (1995). Look-ahead Value Ordering for Constraint Satisfaction Problems. *IJCAI-95*, pp. 572-278.

Gomes, C., C. Fernandez, B. Selman and C. Bessière (2004). Statistical Regimes Across Constrainedness Regions. *10th Conf. on Principles and Practice of Constraint Programming (CP-04)* (M. Wallace, Ed.), pp. 32-46, Springer, Toronto, Canada.

Hulubei, T. and B. O'Sullivan (2005). Search Heuristics and Heavy-Tailed Behavior. *Principles and Practice of Constraint Programming - CP 2005* (P. V. Beek, Ed.), pp. 328-342, Berlin: Springer-Verlag.

Kivinen, J. and M. K. Warmuth (1999). Averaging expert predictions. *Computational Learning Theory: 4th European Conference (EuroCOLT '99)*, pp. 153--167, Springer, Berlin.

Kiziltan, Z., P. Flener and B. Hnich (2001). Towards Inferring Labelling Heuristics for CSP Application Domains. *KI'01*, Springer-Verlag.

Lecoutre, C., F. Boussemart and F. Hemery (2004). Backjump-based techniques versus conflict-directed heuristics. *ICTAI*: 549–557.

Otten, L., M. Grönkvist and D. P. Dubhashi (2006). Randomization in Constraint Programming for Airline Planning. *Principles and Practice of Constraint Programming CP-2006*, pp. 406-420, Nantes, France.

Petrie, K. E. and B. M. Smith (2003). Symmetry breaking in graceful graphs. *Principles and Practice of Constraint Programming CP-2005*, pp. 930--934, LNCS 2833.

Petrovic, S. and S. L. Epstein (2006a). Full Restart Speeds Learning. *Proceedings of the 19th International FLAIRS Conference (FLAIRS-06)*, Melbourne Beach, Florida.

Petrovic, S. and S. L. Epstein (2006b). Relative Support Weight Learning for Constraint Solving. *AAAI Workshop on Learning for Search*, pp. 115-122, Boston.

Petrovic, S. and S. L. Epstein (2007). Random Subsets Support Learning a Mixture of Heuristics. *Proceedings of the 20th International FLAIRS Conference (FLAIRS-07)*, Key West, Florida.

Sabin, D. and E. C. Freuder (1997). Understanding and Improving the MAC Algorithm. *Principles and Practice of Constraint Programming*: 167-181.

Valentini, G. and F. Masulli (2002). Ensembles of learning machines. *Neural Nets WIRN Vietri-02* (M. M. a. R. Tagliaferri, Ed.), Springer-Verlag, Heidelberg, Italy.