# Learning a Mixture of Search Heuristics

Smiljana Petrovic[1], Susan L. Epstein[1,2] and Richard J. Wallace[3]

[1]Department of Computer Science, The Graduate Center, The City University of New York, NY, USA
[2]Department of Computer Science, Hunter College of The City University of New York, NY, USA
[3]Cork Constraint Computation Centre and Department of Computer Science, University College Cork, Cork, Ireland
spetrovic@gc.cuny.edu, susan.epstein@hunter.cuny.edu, r.wallace@4c.ucc.ie

**Abstract.** Problem solvers have at their disposal many heuristics that may support effective search. The efficacy of these heuristics, however, varies with the problem class, and their mutual interactions may not be well understood. The long-term goal of our work is to learn how to select appropriately from among a large body of heuristics, and how to combine them into a weighted mixture that works well on a specific class of problems. During learning, search heuristics' weights are used to solve a problem and then updated based on their subsequent performance. This paper proposes and demonstrates a variety of ways to gauge and adapt search performance, and shows how their application can improve subsequent search performance.

**Keywords:** search, learning, mixture of experts

## 1. Introduction

A program that uses the results of its own search experience to modify its subsequent behavior does *adaptive search*. Such an approach permits the program to tailor its algorithm to the task at hand. In particular, given a set of search heuristics of unknown quality and a class of putatively similar hard problems, our goal is to learn to solve those problems well. The thesis of our work is that adaptive search for a class of constraint satisfaction problems can provide improved performance. This paper describes several techniques that gauge search performance on constraint satisfaction problems and learn to improve it.

Machine learning experiments require both training examples and performance criteria. Given a set of problems, an autonomous learner monitors its performance to direct its own learning. Such a learner has two particular burdens: it must create its own examples and gauge its own performance. A training example here is a search decision of unknown quality. As a result, it is important to gauge the performance of the learner on the particular problem where the example arose. Autonomous learning also requires continuous self-evaluation: Is the program doing well? Has it learned enough? Should it start over? Given a training example, the learning algorithms described here reinforce heuristics that prove successful on a set of problems and discourage those that do not. Our program represents its learned knowledge about how to solve problems as a weighted sum of the output from some subset of its heuristics. Thus this learner's task is both to choose the best heuristics, and to weight them appropriately.

After some basic definitions and a discussion of related work, this paper demonstrates the varied efficacy of individual constraint solving metrics and the potential power available from a mixture of heuristics. It then describes a weighted mixture decision process, explains how our learner extracts training examples from its search experience, and discusses several ways to gauge performance. The paper goes on to detail the experimental design, and to describe some new techniques, each illustrated by an appropriate experiment. Finally, it summarizes our results on effective self-adaptation and plans for future work.

## 2. Background and related work

A constraint satisfaction problem (*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. A *problem class* is a set of CSPs with the same characterization. For example, CSPs in *model B* are characterized by $<n, m, d, t>$, 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) [1]. In a *binary* CSP, each constraint is 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.

Real-world problems typically have non-random structure. A randomly generated problem class may also mandate specific structure for its problems. For example, each of the *composed problems* used here consists of a subgraph (its *central component*) loosely joined to one or more additional subgraphs (its *satellites*) [2].

Traditional CSP search makes a sequence of decisions that instantiates the variables in a problem one at a time with values from their respective domains. After each value assignment, some form of *inference* detects values that are incompatible with the current instantiation. The work reported here uses the MAC-3 algorithm to maintain arc consistency during search [3]. 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 partial instantiation cannot be extended to a solution, so some *retraction* method is applied. Here we use *chronological backtracking*, which prunes the subtree (*digression*) rooted at an inconsistent *node* (assignment of a value to a variable) and withdraws the most recent value assignment(s).

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 [4].

There are only two kinds of search choices here: select a variable or select a value for a variable. Constraint researchers have devised a broad range of variable-ordering and value-ordering heuristics to speed search. Each heuristic relies on its own *metric*, a measure that the heuristic either maximizes or minimizes when it makes a decision. *Min domain* and *max degree* are classic examples of these heuristics. (A full list of the heuristics referenced here appears in the Appendix.) The metric may rely upon dynamic and/or learned knowledge. Each such heuristic may be seen as expressing a preference for choices based on the scores returned by its metric. As demonstrated in the next section, however, no single heuristic is "best." Our research therefore seeks a combination of heuristics.

There are several reasons why a combination of heuristics (known in the mixture literature as *experts*) can offer improved performance, compared to a single expert [5]. 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 [6]. For supervised learning algorithms, the performance of such a *mixture of experts* has been theoretically analyzed in comparison to the best individual expert. 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 [7].

Several learning approaches benefit from a mixture of CSP search heuristics. Multi-TAC selects the best individual heuristic and combines it with each of the others in turn, using the second only as a tiebreaker. In this way Multi-TAC creates several layers of tie-breakers, until no further improvement is obtained [8]. Distributed CSPs can benefit from cooperation and competition of parallel searches led by different heuristics [9]. In contrast, for each search decision the solver described here consults all its chosen heuristics.

**Table 1.** Average number of nodes explored by traditional variable-ordering heuristics (with lexical value ordering) on 50 problems from each of 3 classes. The best performance by a single heuristic (in bold) and the worst (in italics) vary with the problem class.

| ***Heuristic*** | ***<30, 8, 0.26, 0.34>*** | ***<20, 30, 0.444, 0.5>*** | ***<50, 10, 0.38, 0.2>*** |
|---|---|---|---|
| *min domain* | *368.30* | 7,862.68 | 33,270.72 |
| *max degree* | 147.06 | 4,530.72 | *35,978.20* |
| *max forward degree* | 162.96 | *9,123.04* | 35,258.26 |
| *min domain/degree* | 146.96 | 3,145.30 | 22,608.18 |
| *max weighted degree* | 162.76 | 5,139.34 | 24,554.34 |
| *min dom/dynamic deg* | 134.06 | **2,992.56** | 19,947.24 |
| *min dom/weighted deg* | **130.20** | 3,134.70 | **19,608.02** |

## 3. Why learning is necessary

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 in their performance on different classes. Although all the problems used throughout this paper have at least one solution, we control resources with a *step limit* that imposes an upper bound on the number of variable selections and value selections during search on a given problem. Performance is measured in the size of the search tree, that is, the average number of nodes visited during search .

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 heuristics on real-world problems and on problems with non-random structure [10-12]. Consider, for example, a composed problem whose central component is substantially larger, looser (has lower tightness), and sparser (has lower density) than its satellite. Once a solution to the subproblem defined by the satellite is found, it is relatively easy to extend that solution to the looser and sparser central component. In contrast, if one extends a partial solution for the subproblem defined by the central component to the satellite variables, inconsistencies eventually arise deep within the search tree. Typically such problems are either solved with minimal backtracking or go unsolved after hundreds of thousands of steps.

Despite the low density of the central component in such a problem, its variables' degrees are often larger than those in the significantly smaller satellite. This proves particularly challenging for some traditional heuristics. For example, *max degree* (prefer variables with the largest degree in the constraint graph) tends to select variables from the much larger central component first, and therefore fails to solve many such problems within a reasonable step limit. In contrast, the decidedly untraditional *min degree* heuristic tends to prefer variables from the small satellite and thereby detects inconsistencies much earlier. Table 2 shows how traditional heuristics and their duals fare on one class of composed problems. Surprisingly, the duals can do better. We emphasize that the characteristics of such composed problems are often found in real-world problems. To achieve good performance without knowledge about a problem's structure, therefore, it is advisable to consider many popular heuristics along with their duals.

**Table 2.** Average number of nodes explored by 3 traditionally good 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 can perform on problems from this class.

| ***Heuristic*** | ***Percentage solved*** | ***Nodes*** |
|---|---|---|
| *max degree* | *82%* | *14,746.34* |
| min degree | 100% | 34.20 |
| *min domain/degree* | *86%* | *11,882.22* |
| max domain/degree | 92% | 8,409.10 |
| *max weighted degree* | *100%* | *55.22* |
| min weighted degree | 100% | 34.74 |

**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 (in bold).

| *Mixture* | *<30, 8, 0.26, 0.34>* | *<20, 30, 0.444, 0.5>* | *<50, 10, 0.38, 0.2>* |
|---|---|---|---|
| The best representation from Table 1 | 130.2 | 2,992.5 | 19,608.0 |
| *min dom/dynamic degree + max product domain value* | 96.2 | 2,088.4 | 9,721.9 |
| *max weighted degree + max product domain value* | 125.6 | 3,337.0 | 17,449.1 |
| Mixture found by ACE | **90.3** | **1,923.9** | **7,887.1** |

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, two outstanding variable-ordering heuristics from Table 1, *min domain/dynamic degree* and *max weighted degree* benefit differently when coupled with a good value-ordering heuristic, *max product domain value*. The last line of Table 3 demonstrates that a customized combination of more than two heuristics can further improve performance. This paper furthers work on the automatic identification of particularly effective mixtures.

## 4. Solving with a mixture of heuristics

When *ACE* (the Adaptive Constraint Engine) learns to solve a class of CSPs, it customizes a weighted mixture of heuristics for the class [13]. ACE is based on FORR, an architecture for the development of expertise from multiple heuristics [14]. ACE's search algorithm (in Figure 1) alternately selects a variable and then selects a value for it from its domain. The size of the resultant search tree depends upon the order in which values and variables are selected.

Heuristics are implemented by procedures called *Advisors*. ACE's Advisors are organized into three tiers. Tier-1 Advisors make correct decisions without any heuristics. If any of them comments positively on a choice, it is executed. (For example, *Victory* recommends any value from the domain of the last unassigned variable. Since inference has already removed inconsistent values, any remaining value produces a solution.) Tier-2 Advisors address subgoals; they are outside the scope of this paper. The decision-making described here focuses on the Advisors in tier 3. All the tier-3 Advisors are consulted together. As in Figure 1, their output is combined to make a decision by *voting*, which chooses the action with the greatest sum of weighted strengths. Each tier-3 Advisor's heuristic view is based on a descriptive metric. For each metric, there is a dual pair of Advisors, one that favors smaller values for the

```
Search (p, $\mathcal{A}_{var}$, $\mathcal{A}_{val}$)
Until problem p is solved or the allocated resources are exhausted
    Select unvalued variable v
        $v = \arg\max_{c \in V} \sum_{A \in \mathcal{A}_{var}} w(A) \cdot s(A,c)$
    Select value d for variable v from v's domain $D_v$
        $d = \arg\max_{c \in D_v} \sum_{A \in \mathcal{A}_{val}} w(A) \cdot s(A,c)$
    Correct domains of all unvalued variables                *inference*
    Unless domains of all unvalued variables are non-empty
        return to a previous alternative value               *retraction*
```

**Fig. 1.** Search in ACE with a weighted mixture of variable Advisors from $\mathcal{A}_{var}$, and value Advisors from $\mathcal{A}_{val}$. *w(A)* is the weight of Advisor *A; s(A, c)* is the support of Advisor *A* for choice *c.*

metric and one that favors larger values. Typically, only one of them is reported in literature as a heuristic.

## 5. Learning from search experience

Given a class of binary, solvable problems, ACE's goal is to select Advisors and learn weights for them so that the decisions supported by the largest weighted combination of strengths lead to effective search. Our learning scenario specifies that the learner seeks only one solution to one problem at a time, and learns only from problems that it solves. There is no information about whether a single different decision might have produced a far smaller search tree. This is therefore a form of incremental, self-supervised reinforcement learning based only on limited search experience and incomplete information. Moreover, a particular heuristic may be a good choice for some decisions but a poor choice for many others in the same problem.

As a result, any weight-learning algorithm for ACE must select decisions from which to learn, determine what constitutes a heuristic's support for a decision, and specify a way to assign credits and penalties. ACE has two approaches to weight learning: Digression-based Weight Learning (*DWL*) [13] and Relative Support Weight Learning (*RSWL*) [15]. It uses them to update the weights of its tier-3 Advisors.

### From which decisions should one learn?

Both weight-learning algorithms glean training instances from their own (likely imperfect) successful searches. As in Figure 2, *positive training instances* are those made along an error-free path extracted from a solution trace. *Negative training instances* are value selections that led to a digression, as well as variable selections whose subsequent value assignment fails. (Given correct value selections, any vari-

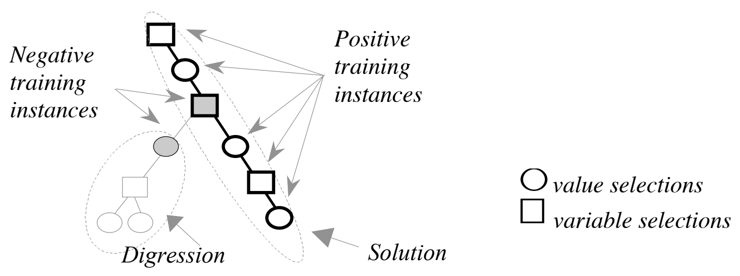

**Fig. 2.** The extraction of positive and negative training instances from the trace of a successful CSP search.

able ordering can produce a backtrack-free solution; we deem a variable selection inadequate if the subsequent value assignment to that variable failed.) Decisions made within a digression do not become training instances.

**What constitutes a heuristic's support of a decision?**
Under DWL, an Advisor is said to support only decisions to which it assigned the highest strength. In contrast, RSWL considers all recommendation strengths. The *relative support* of an Advisor for a choice is the normalized difference between the strength the Advisor assigned to that choice and the average strength it assigned to all available choices. For RSWL, an Advisor supports a choice if its relative support for that choice is positive, and opposes it if its relative support is negative.

**How should one determine credits and penalties?**
As in Figure 3, credits are given to heuristics that support positive training instances and penalties are given to heuristics that support negative training instances. For both DWL and RSWL, an Advisor's weight is the averaged sum of the credits and penal-

***Learn Weights***
```
Until termination of the learning phase
  Identify learning problem $p$
  Search ($p$, $\mathcal{A}_{var}$, $\mathcal{A}_{val}$)
  If $p$ is solved
       for each training instance $t$ from $p$
       for each Advisor $A$ such that $s(A, t) > 0$
              when t is a positive training instance increase $w(A)$        *reward*
              when t is a negative training instance decrease $w(A)$        *penalize*
  else when full restart criteria are satisfied
       initialize all weights to 0.05
```

**Fig. 3.** Learning with random subsets of Advisors from $\mathcal{A}$. The *Search* algorithm is defined in Figure 1.

ties it receives, but the two weight-learning algorithms determine credits and penalties differently.

DWL reinforces Advisors' weights based on the size of the search tree and the size of each digression. An Advisor that supports a positive training instance is rewarded with a weight increment that depends upon the size of the search tree, relative to the minimal size of the search tree in all previous problems. An Advisor that supports a negative training instance is penalized in proportion to the number of search nodes in the resultant digression. Small search trees indicate a good variable order, so the variable-ordering Advisors that support positive training instances from a successful small tree are highly rewarded. For value ordering, however, small search trees are interpreted as an indication that the problem was relatively easy (i.e., any value selection would likely have led to a solution), and therefore result in only small weight increment. In contrast, a successful but large search tree suggests that a problem was relatively difficult, so those value-ordering Advisors that support positive training instances in it receive substantial weight increments [13].

RSWL is more local in nature. With each training instance RSWL reinforces weights based upon the distribution of each heuristic's preferences across all the available choices. RSWL reinforces weights based both upon relative support (defined in Section 5) and upon an estimate of how difficult it is to make the correct decision. For example, an Advisor that strongly singles out the correct decision in a positive training instance receives more credit than a less-discriminating Advisor.

## 6. Experimental design

These learning algorithms are explored in experiments on four classes of randomly generated CSPs. Three are model B classes: <50, 10, 0.38, 0.2>, <20, 30, 0.444, 0.5> and <30, 8, 0.26, 0.34> (abbreviated henceforth as 50-10, 20-30, and 30-8, respectively.) The fourth class, *Comp,* is a class of composed problems, where the central component is 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. All four problem classes are particularly difficult for their size (*n* and *m*). Some of these problems appeared in the First International Constraint Solver Competition at CP-2005.

For ACE, a *learning phase* is a sequence of problems that it attempts to solve and uses to learn Advisor weights. A *testing phase* is a sequence of fresh problems to be solved with learning turned off. A *run* in ACE is a learning phase followed by a testing phase. An *experiment* consists of 10 runs; all data reported here is averaged over 10 runs. Unless it was an experimental parameter, the step limit per problem during learning was 100,000 for 50-10; 10,000 for 20-30; 2,000 for 30-8; and 1,000 for *Comp* problems. Learning terminated after 30 problems, counting from the first solved problem. The step limit per problem during testing was 100,000 for 50-10 and 20-30, and 10,000 for 30-8 and *Comp* problems. For each problem class, each testing phase used the same 50 problems on every run.

In every learning phase, ACE had access to 42 tier-3 Advisors, 28 for variable selection and 14 for value selection (described in the Appendix). Any differences cited are statistically significant at the 95% confidence level.

## 7. Techniques that improve learning

This section describes techniques that use both search performance and problem difficulty to adapt learning. The impact of each technique on our adaptive solver is illustrated empirically.

### 7.1. Full restart

Repeated failure to solve problems can be taken as a reason to restart the entire learning process. If one begins with a large initial list of heuristics that contains minimizing and maximizing versions of many metrics, many of them perform poorly on a particular class of problems (*class-inappropriate heuristics*) while others perform well (*class-appropriate heuristics*). (Note that duals may perform similarly. For example, in Table 2, both *min weighted degree* and *max weighted degrees* are class-appropriate for *Comp*.) On challenging problems, class-inappropriate heuristics occasionally acquire high weights on an initial problem, and then control subsequent decisions, so that either the problems go unsolved or the class-inappropriate heuristics receive additional rewards.

Recall that DWL penalizes in proportion to digression size. As a result, when class-inappropriate heuristics lead to a large digression, a large penalty reduces their impact on decision making. Nonetheless, the learning described here requires that problems be solved, potentially a very expensive process when class-inappropriate heuristics dominate decision making. Recovery from such a situation is faster under *full restart* [16]. Full restart monitors the number of unsolved problems under a reasonably low step limit, and if failures are frequent, the current learning attempt is deemed not promising, the responsible training problems are abandoned, and the entire learning process restarts with freshly initialized weights.

Table 4 illustrates the performance of DWL on 30-8 problems, where the step limit and full restart are treated as parameters. Learning succeeded with a 10,000-step limit, but the total learning cost was high. Reducing the step limit to 2000 or 1000 resulted in an *inadequate run,* one where weights performed poorly and testing performance

**Table 4.** The impact of a step limit and full restart on DWL's performance on 30-8 problems.

| | | Learning | | | | | Testing | |
|---|---|---|---|---|---|---|---|---|
| Step limit | Full restart criterion | Number of problems | Unsolved problems | Nodes | Total learning cost | Full restarts | Nodes | Solved |
| 10000 | none | 30.5 | 3.2 | 1165.2 | 35538.6 | 0.0 | 130.9 | 100.0% |
| 2000 | none | 31.5 | 10.8 | 688.7 | 21695.3 | 0.0 | 707.3 | 94.6% |
| 1000 | none | 31.5 | 8.9 | 334.7 | 10543.1 | 0.0 | 1,318.5 | 87.8% |
| 2000 | 8 in 10 | 42.8 | 10.9 | 541.2 | 23165.1 | 0.8 | 130.3 | 100.0% |
| 1000 | 8 in 10 | 41.1 | 10.2 | 309.0 | 12698.7 | 0.7 | 131.7 | 100.0% |
| 500 | 8 in 10 | 41.1 | 12.3 | 194.6 | 8000.1 | 0.6 | 136.0 | 100.0% |

was unsatisfactory. With full restart, however, even a lower step limit did not have such repercussions — there were no inadequate runs. Although more problems went unsolved during learning under a lower step-limit, those failures were less expensive and the total learning cost decreased. Nonetheless, full restart with a relatively low step limit may demand many learning problems, which are not always available. In the experiments that follow, ACE had recourse to full restart when it failed to solve 8 out of the 10 most recent problems. For brevity, the remaining experiments use RSWL, which responds to full restart similarly.

## 7.2. Learning with random subsets

The interaction among heuristics can also serve as a filter during learning. 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. Rather than consult all its Advisors at once, ACE can randomly select a new subset of Advisors for each problem, consult them, make decisions based on their comments, and update only their weights [17]. This method, *learning with random subsets*, eventually uses a subset in which class-appropriate heuristics predominate and agree on choices that solve a problem.

Figure 4 illustrates how the weights of several heuristics converged during learning with random subsets. Here the problems were drawn from 50-10, and 70% of the Advisors were randomly selected for each problem. Plateaus in weights correspond to problems where the particular heuristic was not selected for the current random subset. Although the weights learned from the first solved problem were not appropriate, during subsequent learning, the class-appropriate heuristics had the opportunity to participate, acquired high weights, and led future decisions. Gradually, ACE separated the class-appropriate heuristics from the class-inappropriate ones. Observe that, as learning progressed, weights stabilized.

In the experiments that follow, for each learning problem, a random number $r$ in [0.3, 0.7] was generated, and then $r$ percent of the variable-ordering Advisors and $r$ percent of the value-ordering Advisors, were selected without replacement to make

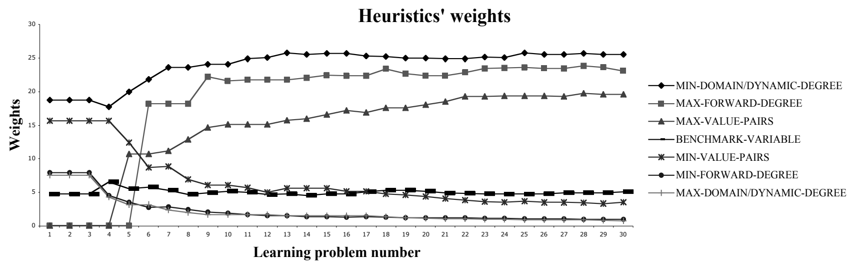

**Fig. 5.** Weights of selected Advisors during learning over 30 problems in a single run.

decisions during search on that problem.

## 7.3. Learning based on decision difficulty

Correct easy decisions are less significant for learning; it is correct difficult decisions that are noteworthy. Thus it may be constructive to estimate the difficulty of each decision the solver faces as if it were a fresh problem, and adjust Advisors' weights accordingly. Our rationale for this is that, on easy problems, any decision leads to a solution. Credit for an easy decision effectively increases the weight of Advisors that support it, but if the decision was made during search, those Advisors probably already had high weights.

The constrainedness parameter $\kappa$ (kappa) has traditionally been used to identify hard classes of problems [18]. For CSPs, κ depends upon *n, d, m,* and *t,* as defined in Section 2:

$$\kappa = \frac{n-1}{2} d \cdot \log_m \left(\frac{1}{1-t}\right)$$

For every search algorithm, and for fixed $n$ and $m$, hard problem classes have κ close to 1. Under one option, however, RSWL uses κ as a measure of subproblem difficulty throughout search. In this case, RSWL rewards only an Advisor that supports the decision in a positive training instance where κ on the problem in the corresponding state is close to 1. Similarly, RSWL penalizes only an Advisor that supports the decision in a negative training instance where κ on the problem in the corresponding state indicates that the subproblem was easy, making errors unacceptable.

Another, and far faster, way to gauge the difficulty of a decision is by the number of choices available. For variable-selection Advisors, the number of available choices is the number of unassigned variables. For value-selection Advisors, the number of available choices is the dynamic domain size of the currently-selected variable. (This number can depend upon the selected inference method.)

Table 5 illustrates that some assessment of problem difficulty can be important in

**Table 5.** A search decision can be viewed as a subproblem whose difficulty serves as input to the RSWL weight-learning algorithm. A statistically significant improvement in search tree size (in bold) occurs only on the 20-30 class.

| | *20-30* | | | | *50-10* | | | |
|---|---|---|---|---|---|---|---|---|
| | ***Learning*** | | ***Testing*** | | ***Learning*** | | ***Testing*** | |
| *Algorithm* | *Tasks* | *Full restarts* | *Nodes* | *Solved* | *Tasks* | *Full restarts* | *Nodes* | *Solved* |
| RSWL | 32.9 | 0.1 | 3,768.4 | 100% | 33.7 | 0.1 | 13,748.6 | 92.2% |
| RSWL with number of choices | 34.6 | 0.2 | **2,286.1** | 100% | 35.4 | 0.1 | 13,185.7 | 91.2% |
| RSWL with κ | 35.1 | 0.1 | **2,022.7** | 100% | 33.1 | 0.0 | 13,109.7 | 94.2% |

adaptive learning. We tested each of the two problem-difficulty strategies in turn. In the second line, penalties were inversely proportional to the number of available choices. In the third line, credits were given only on position training instances when $|\kappa-1| < 0.5$, and penalties were assessed only on a negative training instance when $|\kappa-1| > 0.3$. There was a statistically significant improvement only on the 20-30 class. In the experiments that follow, RSWL's penalties were based on decision difficulty, as measured by the number of available choices.

## 7.4. Selecting heuristics based upon their learned weights

Learned weights can serve as a filter to select appropriate heuristics. Two *benchmark* Advisors, one for value selection and one for variable selection, generate random comments as a lower bound for performance. These benchmarks are excluded from decision making, but used as a baseline to select heuristics for use after learning weights. During testing ACE excludes from its learned mixture any Advisor whose weight is lower than that of its respective benchmark. Further reduction to a pre-specified number of the top-weighted Advisors can produce a significant speedup during testing. Extensive reductions, however, typically result in a significant decline in performance, as we show in our experiments [19].

Table 6 illustrates the impact of selecting heuristics for the testing phase based upon weights learned by RSWL. Under ACE's traditional approach, the benchmark criterion typically eliminates about half the initial Advisors. (In the experiments performed for this paper, 16 out of 28 variable-ordering Advisors and 6 out of 14 value-ordering Advisors usually survived the benchmark criterion on any given run.) The elimination of some surviving Advisors with the lowest weights had no impact on search tree size during testing, but it did provide important reductions in computation time. Since ACE avoids duplicate computations by caching, the computation time required by an Advisor depends not only on its metric, but also on its commonalities with other Advisors. Greater speedup occurred when more value-ordering Advisors were eliminated, because their metrics tend to be more costly. With more extensive reductions, however, search tree size eventually increased. Note that on the composed

**Table 6.** Testing with fewer Advisors. Average search tree size and percent of computational time compared to the traditional (>bmk) approach, which uses every Advisor whose weight is greater than the weight of its benchmark Advisor. Statistically significant differences are in bold.

| | | ***Comp*** | | | ***30-8*** | | ***20-30*** | | ***50-10*** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Var. Adv. | Val. Adv. | *Nodes* | *Solved* | *Time* | *Nodes* | *Time* | *Nodes* | *Time* | *Nodes* | *Solved* | *Time* |
| >bmk | >bmk | 327 | 97% | 100% | 121 | 100% | 2,286 | 100% | 13,186 | 91% | 100% |
| 8 | 4 | 407 | 95% | 95% | 124 | 90% | 2,369 | **71%** | 13,390 | 93% | **84%** |
| 8 | 2 | **492** | 94% | 106% | 124 | **68%** | 2,337 | **55%** | 13,014 | 93% | **70%** |
| 4 | 4 | **755** | 91% | **158%** | **141** | 94% | **2,632** | **87%** | 13,069 | 94% | **77%** |
| 4 | 2 | **661** | 92% | 122% | **141** | **69%** | **2,608** | **67%** | 13,168 | 94% | **69%** |

problems, the extensive increase in number of nodes eventually increases computation time as well. In the experiments that follow, only 8 highly weighted variable Advisors and 2 value Advisors were consulted during testing.

## 7.5. Combining heuristics' preferences

The preferences expressed by heuristics can be used to make decisions during search. The intuition here is that comparative nuances, as expressed by preferences, contain more information than just what is "best." Recall that each heuristic reflects an underlying metric and returns a *score* for each possible choice. Comparative opinions (here, *heuristics' preferences*) can be exploited in a variety of ways that consider both the scores returned by the metrics on which these heuristics rely and the distributions of those scores across a set of possible choices. The simplest way to combine heuristics' preferences is to scale them into some common range. Mere ranking of these scores, however, reflects only the preferences of one choice over another and ignores the degree of metric difference. *Interpolation* not only considers the relative position of choices, but also the actual differences between scores. Linear interpolation makes strength differences proportional to the differences in scores; exponential interpolation emphasizes the top-scoring choices more dramatically. The *Borda methods* were inspired by an election method devised by Jean-Charles de Borda in the 1770s. They emphasize the relative position of a choice among other choices, and attend to how many choices share the same score. The variation used here sets strength equal to the number of choices scored lower than this choice [19].

Table 7 illustrates the impact preference expressions have on learning. These final experiments used all the techniques discussed here: reinforcement depended upon the number of available choices, random subsets of heuristics were consulted for each learning problem, full restarts were performed on unpromising learning attempts, and during testing, ACE used only its eight highest-weighted variable-ordering Advisors and two highest-weighted value-ordering Advisors. No one method to combine preferences outperformed ranking on every problem class, but each was better on some individual classes.

**Table 7.** Performance on different problem classes under a variety of selection regimes. Statistically significant reductions in search tree size are in bold.

| | *Comp* | | 30-8 | | 20-30 | | 50-10 | |
|---|---|---|---|---|---|---|---|---|
| | *Nodes* | *Solved* | *Nodes* | *Solved* | *Nodes* | *Solved* | *Nodes* | *Solved* |
| Ranking | 492.2 | 94.4% | 124.4 | 100% | 2,337.3 | 100% | 13,014.4 | 93.20% |
| Exponential | 615.6 | 92.4% | **108.8** | 100% | 2,145.9 | 100% | 13,440.6 | 93.80% |
| Linear | 696.1 | 92.4% | **94.8** | 100% | 2,295.6 | 100% | **10,095.3** | 95.40% |
| Borda | **255.7** | 97.6% | **111.4** | 100% | 2,494.1 | 100% | **9,348.8** | 95.40% |

## 8. Conclusion and future work

ACE is a successful, adaptive solver. It learns to select a weighted mixture of heuristics for a given problem class that produces search trees smaller than those from outstanding individual heuristics in the CSP literature. ACE learns from its own search performance, and from the accuracy, intensity, frequency and distribution of its heuristics' preferences. ACE adapts its decision making, its reinforcement policy, and its heuristic selection mechanisms effectively.

Our current work extends on several fronts. Rather than rely on an endless set of fresh problems, we plan to reuse unsolved problems and implement boosting with little additional effort during learning [20]. A major focus is the automated selection of good parameter settings for an individual class (including its step limit and full-restart parameters), given [21]. Current work seeks to reduce the degree of randomness in random subset selection as learning progresses. Given the data in Table 7, the ideal number of surviving Advisors clearly depends upon our willingness to trade decision quality for speed during search. We also intend to extend our research to classes containing both solvable and unsolvable problems, and to optimization problems. Finally, we plan to study this approach further in light of the factor analysis evidence for strong correlations between CSP ordering heuristics [22].

## Appendix

Two vertices with an edge between them are *neighbors*. Here, the *degree of an edge* is the sum of the degrees of its endpoints, and the *edge degree of a variable* is the sum of edge degrees of the edges on which it is incident.

**Metrics for variable selection** were static degree, dynamic domain size, FF2 [23], dynamic degree, number of valued neighbors, ratio of dynamic domain size to dynamic degree, ratio of dynamic domain size to degree, number of acceptable constraint pairs, static and dynamic edge degree with preference for the higher or lower degree endpoint, weighted degree, and ratio of dynamic domain size to weighted degree [24]. Each metric produces two Advisors.

**Metrics for value selection** were number of value pairs for the selected variable that include this value, and, for each potential value assignment: minimum resulting domain size among neighbors, number of value pairs from neighbors to their neighbors, number of values among neighbors of neighbors, neighbors' domain size, a weighted function of neighbors' domain size, and the product of the neighbors' domain sizes. Each metric produces two Advisors.

## Bibliography

1. Gomes, C., Fernandez, C., Selman, B., Bessière, C.: Statistical Regimes Across Constrainedness Regions. In: Wallace, M. (ed.): 10th Conf. on Principles and Practice of Constraint Programming (CP-04), Vol. 3258. Springer, Toronto, Canada (2004) 32-46
2. Aardal, K.I., Hoesel, S.P.M.v., Koster, A.M.C.A., Mannino, C., Sassano, A.: Models and

solution techniques for frequency assignment problems. 4OR: A Quarterly Journal of Operations Research **1(4)** (2003) 261-317

3. Sabin, D., Freuder, E.C.: Understanding and Improving the MAC Algorithm. Principles and Practice of Constraint Programming (1997) 167-181
4. Hulubei, T., O'Sullivan, B.: Search Heuristics and Heavy-Tailed Behavior. In: Beek, P.V. (ed.): Principles and Practice of Constraint Programming - CP 2005. Berlin: Springer-Verlag (2005) 328-342
5. Valentini, G., Masulli, F.: Ensembles of learning machines. In: Tagliaferri, M.M.a.R. (ed.): Neural Nets WIRN Vietri-02. Springer-Verlag, Heidelberg, Italy (2002)
6. Dietterich, T.G.: Ensemble methods in machine learning. First International Workshop on Multiple Classifier Systems, Cagliari, Italy (2000) 1-15
7. Kivinen, J., Warmuth, M.K.: Averaging expert predictions. Computational Learning Theory: 4th European Conference (EuroCOLT '99). Springer, Berlin (1999) 153--167
8. Minton, S., Allen, J.A., Wolfe, S., Philpot, A.: An Overview of Learning in the Multi-TAC System. First International Joint Workshop on Artificial Intelligence and Operations Research, Timberline, Oregon, USA (1995)
9. Ringwelski, G., Hamadi, Y.: Boosting Distributed Constraint Satisfaction. Principles and Practice of Constraint Programming CP-2005 (2005) 549-562
10. Lecoutre, C., Boussemart, F., Hemery, F.: Backjump-based techniques versus conflict directed heuristics. ICTAI (2004) 549–557
11. Otten, L., Grönkvist, M., Dubhashi, D.P.: Randomization in Constraint Programming for Airline Planning. Principles and Practice of Constraint Programming CP-2006, Nantes, France (2006) 406-420
12. Petrie, K.E., Smith, B.M.: Symmetry breaking in graceful graphs. Principles and Practice of Constraint Programming CP-2005. LNCS 2833 (2003) 930-934
13. Epstein, S.L., Freuder, E.C., Wallace, R.: Learning to Support Constraint Programmers. Computational Intelligence **21(4)** (2005) 337-371
14. Epstein, S.L.: For the Right Reasons: The FORR Architecture for Learning in a Skill Domain. Cognitive Science **18** (1994) 479-511
15. Petrovic, S., Epstein, S.L.: Relative Support Weight Learning for Constraint Solving. AAAI Workshop on Learning for Search, Boston (2006) 115-122
16. Petrovic, S., Epstein, S.L.: Full Restart Speeds Learning. Proceedings of the 19th International FLAIRS Conference (FLAIRS-06), Melbourne Beach, Florida (2006)
17. Petrovic, S., Epstein, S.L.: Random Subsets Support Learning a Mixture of Heuristics. 20th International FLAIRS Conference (FLAIRS-07), Key West, Florida (2007)
18. Gent, I.P., Prosser, P., Walsh, T.: The Constrainedness of Search. AAAI/IAAI **1** (1996) 246-252
19. Petrovic, S., Epstein, S.L.: Preferences Improve Learning to Solve Constraint Problems. AAAI Workshop on Preference Handling for Artificial Intelligence, Vancouver, Canada (2007) 71-78
20. Schapire, R.E.: The strength of weak learnability. Machine Learning **5(2)** (1990) 197--227
21. Hutter, F., Hamadi, Y., Hoos, H.H., Leyton-Brown, K.: Performance Prediction and Automated Tuning of Randomized and Parametric Algorithms. Principles and Practice of Constraint Programming CP-2006 (2006) 213-228
22. Wallace, R.J.: Factor analytic studies of CSP heuristics. In: Beek, P.v. (ed.): Principles and Practice of Constraint Programming - CP 2005, Vol. 3709. Springer (2005) 712-726
23. Smith, B., Grant, S.: Trying Harder to Fail First. European Conference on Artificial Intelligence (1998) 249-253
24. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting Systematic Search by Weighting Constraints. ECAI 2004 (2004) 146-150