Constraint Programming Letters 4 (2009) 15-38 Submitted 6/2008; Published 12/2008

# Tailoring a Mixture of Search Heuristics

**Smiljana V. Petrovic** SPETROVIC@IONA.EDU
*Department of Computer Science*
*Iona College*
*New Rochelle, NY, USA*

**Susan L. Epstein** SUSAN.EPSTEIN@HUNTER.CUNY.EDU
*Department of Computer Science*
*Hunter College and The Graduate School of*
*The City University of New York, NY, USA*

**Editors:** Youssef Hamadi, Eric Monfroy, and Frédéric Saubion

**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 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.

## 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. The principal results of this paper gauge search performance on constraint satisfaction problems and learn to improve it by learning with full restart, learning with random subsets of heuristics, learning based on decision difficulty, and the expression of heuristic preferences.

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 discard 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 the 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 describes how our system works with heuristics. Next it 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, and explains how our learner extracts training examples and learns weights based on its search experience. The paper goes on to describe four new techniques to manage a large body of conflicting heuristics. Each technique is then illustrated by an appropriate experiment. The final section summarizes our results on effective self-adaptation and our plans for future work.

## 2. Background

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. In a *binary* CSP, each constraint is on at most two 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, binary CSPs in *model B* are characterized by $\langle n, m, d, t \rangle$, where $n$ is the number of variables, $m$ the maximum domain size, $d$ the density (fraction of constraints out of $n(n\text{-}1)/2$ possible constraints) and $t$ the *tightness* (fraction of possible value pairs that each constraint excludes) (Gomes, et al., 2004). 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*) (Aardal, et al., 2003). Figure 1 illustrates a composed problem with two satellites.

A random geometric graph $\langle n, D \rangle$ has $n$ vertices, each represented by a random point in the unit square (Johnson, et al., 1989). There is an edge between two vertices if and only if their (Euclidean) distance is no larger than $D$. A class of random geometric CSPs $\langle n, D, d, t \rangle$ is based on a set of random geometric graphs $\langle n, D \rangle$. In $\langle n, D, d, t \rangle$, the variables represent random

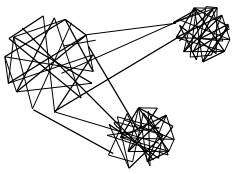


**Figure 1:** A composed problem with two satellites.

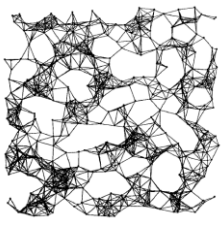


**Figure 2:** A geometric graph from (Johnson, et al., 1989).

points, and constraints are on variables corresponding to points close to each other. Additional edges ensure that the graph is connected. Density and tightness are given by the parameters *d* and *t*, respectively. Figure 2 illustrates a geometric graph with 500 variables.

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 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 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 values to some subset of the variables) and withdraws the most recent value assignment(s).

The efficacy of a constraint solver is gauged by its ability to find a solution to a problem (or prove that none exists), along with the computational resources (CPU time and space in nodes) required to do so. Search for a CSP solution is NP-complete; 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).

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 metrics for the heuristics used in these experiments appears in the Appendix.) A 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 Section 5.1, however, no single heuristic is "best" on all CSP classes. Our research therefore seeks a combination of heuristics.

## 3. Related work

The idea that the combined recommendations of multiple human experts can outperform a single expert goes back at least to the Marquis de Condorcet (1745-1794) (Young, 1988). His Jury Theorem asserts that the judgment of a committee of competent experts, each of whom is correct with probability greater than 0.5, is superior to the judgment of any individual expert. Dietterich gives several reasons for preferring a mixture of hypotheses in machine learning (Dietterich, 2000). On limited data, there may be different hypotheses that appear equally accurate. In this case, although one could approximate the unknown true hypothesis by the simplest one, averaging or mixing all of them together can produce a better approximation. Moreover, even if the target function is not representable by any individual hypothesis in the pool, their combination could produce an acceptable representation.

An *ensemble of classifiers* is a set of classifiers whose individual decisions are combined to classify new examples. *Ensemble methods* create ensemble classifiers. The most popular such algorithm, AdaBoost, seeks classifiers that are better on examples for which the current ensemble's performance is poor (Freund and Schapire, 1996; Schapire, 1990). The solver described here, however, works with prespecified heuristics, and learns how to select and combine them.

A *mixture of experts algorithm* learns, from a sequence of trials, how to combine experts' predictions (Kivinen and Warmuth, 1999). In a supervised environment, a trial has three steps: the mixture algorithm receives predictions from each of *e* experts, makes its own prediction *y* based on them, and then receives the correct value *y´*. The objective is to create a mixture algorithm that minimizes the *loss function* (the distance between *y* and *y´*). The performance of such an algorithm is often measured by its *relative loss*: the additional loss compared to the best individual expert. Under the worst-case assumption, mixture of experts algorithms have been proved asymptotically close to the behavior of the best expert (Kivinen and Warmuth, 1999). The learning described here, however, does not have external supervision; the training instances come from the solver's own (likely imperfect) successful searches. As a result, learning lacks reliable information on how much any individual decision or the mutual interaction of heuristics influenced overall search performance.

There is substantial theoretical and experimental confirmation for the average case performance superiority of mixtures of classifiers, particularly for decision trees and neural networks (Ali and Pazzani, 1996; Opitz and Shavlik, 1996; Valentini and Masulli, 2002). For sufficiently accurate and diverse classifiers, the accuracy of an ensemble increases with the number of classifiers combined (Hansen and Salamon, 1990).

Different algorithms can be combined to solve CSPs. Under *REBA* (Reduced Exceptional Behavior Algorithm), more complex algorithms are applied only to harder problems (Borrett, et al., 1996). REBA begins search with a simple algorithm, and when there is insufficient progress, switches to a more complex algorithm. If necessary, this process can continue through a prespecified sequence of complex algorithms. The ranking can be tailored for a given class of problems, and is usually based on the median cost of solution and an algorithm's sensitivity to exceptionally hard problems from the class. In general these algorithms have better worst-case performance, but a higher average cost when applied to classes with many easy problems that could be quickly solved by simpler algorithms.

*Algorithm portfolios* select a subset of the available algorithms according to some schedule. That subset is run in parallel (or interleaved on a single processor) with evenly distributed resources, until the fastest one solves the problem (Gomes and Selman, 2001). Schedule selection can also allocate additional CPU time to more promising heuristics (*dynamic algorithm portfolios)* (Gagliolo and Schmidhuber, 2006) or improve average-case running time relative to the fastest individual solver (Streeter, et al., 2007). Distributed CSPs can benefit from cooperation and competition during parallel searches led by different heuristics (Ringwelski and Hamadi, 2005).

Selecting among different variable-ordering heuristics has been studied for local search. In one approach, a set of heuristics is associated with each constraint (Nareyek, 2004). Each iteration of local search selects a constraint to be adjusted based on some measure of its inconsistency in the current instantiation. The heuristic that makes an adjustment is selected probabilistically, based on its expected benefit (its *utility value*). All utility values are initially equal, and positively or negatively reinforced based on the difference between current total cost and the total cost the last time that constraint was selected.

*CLASS* discovers good variable-ordering heuristics with a genetic algorithm (Fukunaga, 2002). It begins with a population of randomly generated heuristics, picks two heuristics with probability proportional to some objective function and generates a set of children that are then inserted into the population. Each child replaces a randomly selected member of the population, so that the population remains constant in size. The best heuristic found during the course of the search is returned.

Multi-TAC first generates heuristics specifically for given problems, and then orders them in a list (Minton, et al., 1995). It starts with an initially empty list as a parent. Each of the remaining heuristics, attached to the parent one at a time, creates a child. The *utility* of a child is the number of instances solved within a given time limit for each instance, with total time as a tiebreaker. The child with the highest utility is declared the new parent. The process continues recursively until there is no child better than its parent, or all candidates are exhausted. The heuristics in the resulting list are consulted one by one, moving to the next only as a tiebreaker.

The work reported here differs from these approaches in several important ways. It does not create classifiers as Multi-TAC and classic ensembles do; its heuristics are prespecified. Our solver is self-supervised, without the external performance standard on which the traditional mixture of experts algorithms rely. Our solver is for global search only and does not employ a genetic algorithm, although it does do a kind of reinforcement learning. Whereas REBA and algorithm portfolios combine heuristics according to some predefined instructions, our solver learns to combine them as it tries to solve problems. Finally, only ACE consults all its chosen heuristics on every search decision, and intentionally includes duals (described in Section 5.1) to ensure diversity.

## 4. Solving with a mixture of heuristics

When *ACE* (the Adaptive Constraint Engine) learns to solve a class of binary CSPs, it customizes a weighted mixture of heuristics for the class (Epstein, et al., 2005). ACE is based on FORR, an architecture for the development of expertise from multiple heuristics (Epstein, 1994). ACE's search algorithm (in Figure 3) 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.

***Search (p, $A_{var}$, $A_{val}$)***
Until problem *p* is solved or the allocated resources are exhausted
Select unvalued variable *v*

$$v = \arg\max_{c_{\text{var}} \in V} \sum_{A \in \mathbf{A}_{var}} q(A) \cdot w(A) \cdot s(A, c_{\text{var}})$$

Select value *d* for variable *v* from *v*'s domain $D_v$

$$d = \arg\max_{c_{val} \in D_v} \sum_{A \in \mathbf{A}_{val}} q(A) \cdot w(A) \cdot s(A, c_{val})$$

Correct domains of all unvalued variables **inference**
Unless domains of all unvalued variables are non-empty
return to a previous alternative value **retraction**

**Figure 3:** Search in ACE with a weighted mixture of variable-ordering Advisors from $A_{var}$, and value-ordering Advisors from $A_{val}$. *q(A)* is the discount factor. *w(A)* is the weight of Advisor *A*. *s(A, c)* is the strength of Advisor *A* for choice *c*.

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 final 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 and not used in the experiments reported here.

The decision making described here focuses on the Advisors in tier 3. Each tier-3 Advisor *comments* upon (produces a strength for) some of its favored choices, those whose metric scores are among the *f* most favored. Because a metric can return identical values for different choices, an Advisor usually makes many more than *f* comments. (Here $f = 5$, unless otherwise stated.) The *strength s(A, c)* is the degree of support from Advisor *A* for choice *c.* Each tier-3 Advisor's view is based on a descriptive metric. All tier-3 Advisors are consulted together. As in Figure 3, a decision in tier 3 is made by *weighted voting*, where the strength $s(A, c)$ given to choice *c* by Advisor *A* is multiplied by the *weight* $w(A)$ of Advisor *A.* All weights are initialized to 0.05, and then learned for a class of problems by the process described in the next section. The discount factor *q(A)* in (0,1] modulates the influence of Advisor *A* until it has commented often enough during learning. As data is observed on *A*, *q(A)* moves toward 1, effectively increasing the impact of *A* on a given class as its learned weight becomes more trustworthy.

Weighted voting selects the choice with the greatest sum of weighted strengths from all Advisors. (Ties are broken randomly.) Each tier-3 Advisor's heuristic view is based on a descriptive metric. For each metric, there is a pair of Advisors, one that favors smaller values for the metric and one that favors larger values. Typically, only one of the pair has been reported in the literature as a heuristic.

## 5. Learning from search experience

The motivation for this work and the experiments in responses to them are demonstrated here on two classes of structured problems (geometric and composed problems) and four model B classes. *Geo* is the class of geometric problems <50, 10, 0.4, 0.82>. *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. Some of these problems appeared in the First International Constraint Solver Competition at CP-2005. All problems are randomly generated solvable binary CSPs. The additional model B classes are <50, 10, 0.38, 0.2>, which is exceptionally hard for its *size* ($n$ and $m$); <50, 10, 0.18, 0.37>, which is the same size but somewhat easier; <20, 30, 0.444, 0.5>, whose problems have large domains; and <30, 8, 0.26, 0.34> which are easy compared to the other classes, but difficult for their size.

Resources are controlled here with a *node limit* that imposes an upper bound on the number of assignments of a value to a variable during search on a given problem. Unless otherwise noted, the node limit per problem was 50,000 for <50, 10, 0.38, 0.2>; 20,000 for <20, 30, 0.444, 0.5>; 10,000 for <50, 10, 0.18, 0.37>; 500 for <30, 8, 0.26, 0.34>; and 5,000 for *Comp* and *Geo* problems.

### 5.1 Why learning is necessary

Learning is necessary for CSP solution because even well-trusted individual heuristics vary dramatically in their performance on different classes. For example, in Table 1 the Min-domain/dynamic-degree heuristic is the most successful on <20, 30, 0.444, 0.5> problems, but it is inadequate on *Comp* problems (A heuristic's performance was declared *inadequate* if it failed to solve 10 out of 50 problems under a specified resource limit.)

| *Advisors* | *Geo* | | *Comp* | | *<20, 30, 0.444, 0.5>* | |
|---|---|---|---|---|---|---|
| | ***Nodes*** | ***Solved*** | ***Nodes*** | ***Solved*** | ***Nodes*** | ***Solved*** |
| Min-domain/dynamic-degree | 258.1 | 98% | *inadequate* | – | **3403.4** | 100% |
| Min-dynamic-domain/weighted-degree | **246.4** | 100% | 57.67 | 100% | 3534.3 | 100% |
| Min-domain/static-degree | 254.6 | 98% | *inadequate* | – | 3561.3 | 100% |
| Max-static-degree | *397.7* | 98% | *inadequate* | – | 4742.1 | 96% |
| Max-weighted-degree | 343.3 | 98% | **50.44** | 100% | *5827.9* | 98% |

**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.

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 (Lecoutre, et al., 2004; Otten, et al., 2006; Petrie and Smith, 2003). Consider, for example, a *Comp* 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. A typical CSP solver either solves such a problems with minimal backtracking or exhausts its resources after hundreds of thousands of nodes. 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.

The selection of appropriate heuristics from the many touted in the constraint literature is non-trivial. For example, *max static degree* tends to select variables from the much larger central component first, and therefore fails to solve many such problems within a reasonable node limit. In contrast, the decidedly untraditional *min static degree* heuristic tends to prefer variables from the small satellite and thereby detects inconsistencies much earlier. Table 2 shows how three traditional heuristics and their duals fare on one class of composed problems. Two of the duals do better than traditionally good heuristics, but that is not always the case. We emphasize again 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.

| ***Heuristic*** | ***Nodes*** | ***Solved*** |
|---|---|---|
| Min-static-degree | 33.15 | 100% |
| *Max-static-degree* | inadequate | – |
| Max-domain/dynamic-degree | 532.22 | 95% |
| *Min-domain/dynamic-degree* | inadequate | – |
| Max-domain | 1168.71 | 90% |
| *Min-domain* | 373.22 | 97% |

**Table 2:** Average number of nodes explored by three traditional heuristics (in italics) and their duals on *Comp* problems (described in the text). Note the better performance of two of the duals here.

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 we tested. The second line illustrates the performance of a random selection of heuristics, without any learning. On one class, they proved inadequate on every run, and on the other class, five runs were inadequate while the other five dramatically underperformed every other approach. The third line shows 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 the best such pair is not trivial.) The last line of Table 3 demonstrates that a customized combination of more than two heuristics, discovered with the methods described here, can further improve performance. This paper furthers work on the automatic identification of particularly effective mixtures.

| | ***<50, 10, 0.38, 0.2>*** | | ***<20, 30, 0.444, 0.5>*** | |
|---|---|---|---|---|
| ***Guidance*** | ***Nodes*** | ***Solved*** | ***Nodes*** | ***Solved*** |
| Best individual heuristic tested | 17,399.06 | 84.00% | 3,403.42 | 100.00% |
| Randomly selected combination of more than 2 heuristics | 10 inadequate runs | — | 5 inadequate runs | — |
| Best pair of variable-ordering and value-ordering heuristic identified | 10,889.00 | 96.00% | 1,988.10 | 100.00% |
| Best learned weighted combination of more than 2 heuristics found by ACE | 8,559.66 | 98.00% | 1,956.62 | 100.00% |

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

### 5.2 How ACE learns Advisors' weights

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*) (Epstein, et al., 2005) and Relative Support Weight Learning (*RSWL*) (Petrovic and Epstein, 2006b). It uses them to update the weights of its tier-3 Advisors.

#### Decisions from which to learn

Both weight-learning algorithms glean training instances from their own (likely imperfect) successful searches. As in Figure 4, *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 failed. (Given correct value selections, any variable ordering can produce a backtrack-free solution; we deem a variable selection inadequate if the subsequent value assignment to that variable failed.) Decisions below the root of a digression do not become training instances.

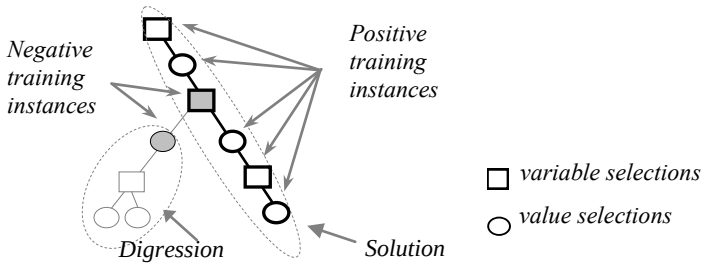



**Figure 4:** The extraction of positive and negative training instances from the trace of a successful CSP search.

## How weights are adjusted

Under DWL, an Advisor is said to support only those decisions to which it assigned the highest strength. In contrast, RSWL considers all 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 at that decision point. For RSWL, an Advisor supports a choice if its relative support for that choice is positive, and opposes that choice if its relative support is negative.

As in Figure 5, heuristics that support positive training instances receive credits, and heuristics that support negative training instances receive penalties. For both DWL and RSWL, an Advisor's weight is the averaged sum of the credits and penalties it receives, but the two weight-learning algorithms determine credits and penalties differently.

***Learn Weights***
Initialize all weights to 0.05
Until termination of the learning phase
  Identify learning problem *p*
  *Search* (*p*, $A_{var}$, $A_{val}$)
  If *p* is solved
    then for each training instance *t* from *p*
      for each Advisor *A* that supports *t*
      when t is a positive training instance, increase *w*(*A*) **credit**
      when t is a negative training instance, decrease *w*(*A*) **penalize**
    else when full restart criteria are satisfied
      initialize all weights to 0.05

**Figure 5:** Learning weights for Advisors. The *Search* algorithm is defined in Figure 3.

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, a small search tree is interpreted as an indication that the problem was relatively easy (i.e., any value selection would likely have led to a solution), and therefore results in only small weight increments. 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 from it receive substantial weight increments (Epstein, et al., 2005).

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 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, and the penalty for a wrong choice from among a few is harsher then for a wrong choice from among many.

In addition to an input set of Advisors, ACE has one benchmark for variable ordering and another for value ordering. Each *benchmark Advisor* models random advice; it makes random comments with random strengths. Although the benchmarks' comments never participate in decision making, the benchmarks themselves earn weights. That weight serves as a filter for the benchmark's associated Advisors; an Advisor must have a learned weight higher than its benchmark's (i.e., provide better than random advice) to be constructive.

## 6. Techniques that improve learning

This section describes four techniques that use both search performance and problem difficulty to adapt learning.

### 6.1 Full restart

Repeated failure to solve problems motivates restarting the entire learning process. If one begins with a large initial list of heuristics that contains minimizing and maximizing versions of many metrics, some perform poorly on a particular class of problems (*class-inappropriate heuristics*) while others perform well (*class-appropriate heuristics*). On challenging problems, class-inappropriate heuristics occasionally acquire high weights on an initial problem, and then control subsequent decisions. As a result, subsequent problems may have extremely large search trees.

Under unlimited resources, DWL will recover from class-inappropriate heuristics with high weights, because they typically generate large search trees and large digressions. In response, DWL will impose large penalties and provide small credits to the variable-ordering Advisors that lead decisions. With their significantly reduced weights, the class-inappropriate Advisors will no longer dominate the class-appropriate Advisors. Nonetheless, solving a hard problem without good heuristics is computationally expensive. If adequate resources are unavailable under a given node limit and a problem goes unsolved, no weight changes occur at all.

*Full restart* monitors the frequency and the order of unsolved problems in the problem sequence. If it deems the current learning attempt not promising, full restart abandons the learning process (and any learned weights) and begins learning on new problems with freshly initialized weights (Petrovic and Epstein, 2006a). The node limit is a critical parameter for full restart. Because ACE abandons a problem if it does not find a solution within the node limit, the node limit is the criterion for unsuccessful search. Since the full restart threshold directly depends upon the number of failures, the node limit is the performance standard for full restart. The node limit also controls resources; lengthy searches permitted under high node limits are expensive.

With higher node limits, weights can eventually recover without the use of full restart, but recovery is more expensive. With lower node limits, the cost of learning (total number of nodes across all learning problems) with full restart is slightly higher than without it. The learner fails on all the difficult problems, and even on some of medium difficulty, repeatedly triggering full restart until the weight profile is good enough to solve almost all the problems. Full restart abandons some problems and uses additional problems, which increases the cost of learning. The difference in cost is small, however, since each problem's cost is limited under a relatively low node limit. As the node limit increases, full restart is able to avoid inadequate runs, but at a higher cost. It takes longer to trigger full restart because the learned weight profile is good enough, so that failures are less frequent. Moreover, with a high node limit, every failure is expensive. When full restart eventually triggers, the prospect of relatively extensive effort on further problems is gone. Because it detects and eliminates unpromising learning runs early, full restart avoids many costly searches and drastically reduces overall learning cost. Experimental results and further discussion of full restart appear in Section 7.1.

### 6.2 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 any problem within a given node 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 (Petrovic and Epstein, 2008). This method, *learning with random subsets*, eventually uses a subset in which class-appropriate heuristics predominate and agree on choices that solve a problem.

For a fixed node limit and set of heuristics, an underlying assumption here is that the ratio of class-appropriate to class-inappropriate heuristics determines whether a problem is likely to be solved. When class-inappropriate heuristics predominate in a set of heuristics, the problem is unlikely to be solved and no learning occurs. The selection of a new random subset of heuristics for each new problem, however, should eventually produce some subset $S$ with a majority of class-appropriate heuristics that solves its problem within a reasonable resource limit. As a result, the Advisors in $S$ will have their weights adjusted. On the next problem, the new random subset $S'$ is likely to contain some low-weight Advisors outside of $S$, and some reselected from $S$. Any previously-successful Advisors from $S$ that are selected for $S'$ will have larger positive weights than the other Advisors in $S'$, and will therefore heavily influence search decisions. If $S$ succeeded because it contained more class-appropriate than class-inappropriate heuristics, $S \cap S'$ is also likely to have more class-appropriate heuristics and therefore solve the new problem, so again those that participate in correct decisions will be rewarded. On the other hand, in the less likely

case that the majority of $S \cap S'$ consists of reinforced, class-inappropriate heuristics, the problem will likely go unsolved, and the class-inappropriate heuristics will not be rewarded further.

Learning with random subsets manages a substantial set of heuristics, most of which may be class-inappropriate and contradictory. It results in fewer *early failures* (problems that go unsolved under initial weights, before any learning occurs) within the given node limit, and thereby makes more training instances available for learning. Learning with random subsets is also expedited by faster decisions during learning because it solicits less advice.

When there are roughly as many class-appropriate as class-inappropriate Advisors, the subset sizes are less important than when class-inappropriate Advisors outnumber class-inappropriate ones. Intuitively, if there are few class-appropriate heuristics available, the probability that they are selected as a majority in a larger subset is small (indeed, 0 if the subset size is more than twice the number of class-appropriate Advisors). For example, given $a$ class-appropriate Advisors, and $b$ class-inappropriate Advisors, the probability that the majority of $r$ randomly-selected Advisors is class-appropriate is

$$p = \sum_{k=\left\lfloor \frac{r}{2}+1 \right\rfloor}^{r} \frac{\binom{a}{k}\binom{b}{r-k}}{\binom{a+b}{r}} \qquad [1]$$

and the expected number of trials until the subset has a majority of class-appropriate Advisors is

$$\sum_{i=1}^{\infty} i(1-p)^{i-1} p = \frac{1}{p} \qquad [2]$$

When there are more class-inappropriate Advisors ($a < b$), a smaller set is more likely to have a majority of class-appropriate Advisors. For example, if $a = 6$, $b = 9$, and $r = 4$, [1] evaluates to 0.14 and [2] to 7. For $a = 6$, $b = 9$, and $r = 10$, however, the probability of a randomly selected subset with a majority of class-appropriate heuristics is only 0.04 and the expected number of trials until the subset has a majority of class-appropriate Advisors is 23.8.

Weights converge faster when subsets are larger. When the random subsets are smaller, subsequent random subsets are less likely to overlap with those that preceded them, and therefore less likely to include Advisors whose weights have been revised. As a result, failures occur often, even after some class-appropriate heuristics receive high weights. ACE monitors its learning progress, and adapts the size of random subsets. Initially, random subsets are small, but as learning progresses and more Advisors participate and obtain weights, the size of the random subsets increase. This makes overlap more likely, and thereby speeds learning. Experimental results and further discussion of learning with random subsets appear in Section 7.2.

### 6.3 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. We have addressed this issue with two algorithms, each dependent upon a single parameter.

The constrainedness parameter $\kappa$ (kappa) has traditionally been used to identify hard classes of problems (Gent, et al., 1996). For CSPs, $\kappa$ depends upon $n$, $d$, $m$, and $t$, as defined in Section 2:

$$\kappa = \frac{n-1}{2} d \cdot \log_m(\frac{1}{1-t}) \quad [3]$$

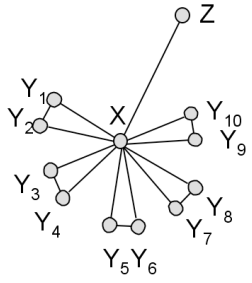
For every search algorithm, and for fixed *n* and *m*, hard problem classes have κ close to 1. Under one option, however, RSWL uses κ to measure the difficulty $\kappa_P$ of subproblem *P* at each decision point. *RSWL-κ* depends on a parameter *k*. It gives credit to an Advisor only when that Advisor supports a positive training instance derived from a search state where $|\kappa_P - 1| < k$. RSWL-κ penalizes an Advisor only when it supports a negative training instance derived from a search state where $|\kappa_P - 1| > k$.

Because calculating $\kappa_P$ on every training instance is computationally expensive, another variant, the *RSWL-d* algorithm, uses the number of unassigned variables at the current search node as a rough, quick estimate of problem hardness. Decisions at the top of the search tree are known to be more difficult (Ruan, et al., 2004). For a given parameter *h*, no penalty is given at all for any decision in the top *h* percent of the nodes in the search tree, and no credit is given for any decision below them. Experimental results and further discussion of learning with decision difficulty appear in Section 7.3.

### 6.4 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 that 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, not the extent to which one choice is preferred over another. For example in Figure 6, the degrees of variables *X* and $Y_1$ differ by 9, while the degrees of $Y_1$ and *Z* differ by only 1. Nonetheless, ranking assigns equally spaced strengths (3, 2 and 1, respectively) to those variables. Ranking also ignores how many choices share the same score. For example in Figure 6, the ranks of choices $Y_1$ and *Z* differ by only 1, although the heuristic prefers only one choice over $Y_1$ and 11 choices over *Z*. We have explored several methods that express Advisors preferences and address those shortcomings (Petrovic and Epstein, 2007).




| ***Variables*** | ***X*** | ***$Y_1$ - $Y_{10}$*** | ***Z*** |
|---|---|---|---|
| **Degree metric scores** | **11** | **2** | **1** |
| Rank strength | 3 | 2 | 1 |
| Linear strength | 3.00 | 1.20 | 1.00 |
| *Borda-w* strength | 2.83 | 1.17 | 1.00 |
| *Borda-wt* strength | 3.00 | 2.83 | 1.17 |

**Figure 6:** A constraint graph for a CSP problem on 12 variables, and the impact of different preference expression methods on a single metric.

*Linear interpolation* not only considers the relative position of scores, but also the actual differences between them. Under linear interpolation, strength differences are proportional to score differences. For example, in Figure 6, strengths can be determined by the value of the linear function through the points (11, 3) and (1, 1). Instead of strength 2 for all the *Y* variables, linear interpolation gives them strength 1.2, which is closer to the strength 1 given to variable *Z*, because the degrees of the *Y* variables are closer to the degree of *Z*. The significantly higher degree of variable *X* is reflected in the distance between its strength and those given to the other variables.

The *Borda methods* were inspired by an election method devised by Jean-Charles de Borda in the late eighteenth century (Saari, 1994). Borda methods consider the total number of available choices, the number of choices with a smaller score and the number of choices with an equal score. Thus the strength for a choice is based on its position relative to the other choices. To keep strengths in the same range as those from ranking and linear interpolation, Borda methods normalize by accumulating *points* (the ratio of the rank strength range to the number of choices on which an Advisor comments). For example, in Figure 6, a point is (3–1)/12=0.17.

The first Borda method, *Borda-w*, awards a point for each *win* (metric score higher than the score of some other commented choice). Examples for *Borda-w* strengths are also shown in Figure 6. The set of lowest-scoring variables (here, only *Z*) always has strength 1. Because every *Y* variable out-scored only *Z*, the strength of any *Y* variable is 1+0.17=1.17. The highest-scoring choice *X* out-scored 11 choices, so *X*'s strength is $1+11 \cdot 0.17=2.83$.

The second Borda method, *Borda-wt*, awards one point for each win and one point for each *tie* (score equal to the score of some other choice). It can be interpreted as emphasizing losses. The highest-scoring set of variables (here, only *X*) always has strength equal to the number of subsets of variables to be scored. For example, in Figure 6, no choice out-scored the highest-scoring choice *X*, so its strength is 3, one choice (*X*) outscored the *Y* variables, so their strengths are reduced by one point (3–0.17=2.83), and 11 choices out-scored *Z*, resulting in strength 3–11*0.17=1.17.

The difference between the two Borda methods is evident when many choices share the same score. *Borda-w* 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. Under *Borda-wt*, a large subset results in a big gap in strength between that subset and the next (less preferred) one. In Figure 6, for example, the 10 *Y* variables share the same score. Under *Borda-w*, the difference between the strength of any *Y* variable and *X* is 1.66, while the difference between the strength of any *Y* variable and *Z* is only 0.17. Under *Borda-wt*, however, the difference between the strength of any *Y* and *X* is only 0.17, while the difference between the strength of any *Y* and *Z* is 1.66. Experimental results and further discussion of learning with preferences appear in Section 7.4.

## 7. Results

The methods in the previous section are investigated here with ACE. 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. Each *experiment* reported here averages its results over 10 runs. Any differences cited are statistically significant at the 95% confidence level.

In the experiments that follow, learning terminated after 30 problems, counting from the first solved problem, or if no problem in the first 30 was solved. Under full restart, more learning problems can be used, but the upper bound for the total number of problems in a learning phase is

always 80. For each problem class, every testing phase used the same 50 problems. When any 10 of the 50 testing problems went unsolved within the node limit, learning in that run was declared *inadequate* and further testing was halted. In every learning phase, ACE had access to 40 tier-3 Advisors, 28 for variable selection and 12 for value selection (described in the Appendix). During a testing phase, ACE uses only those Advisors whose learned weights exceeded that of their respective benchmarks.

### 7.1 Full restart improves performance

The benefits of full restart are illustrated here on <30, 8, 0.26, 0.34> problems with DWL, where the node limit during learning is treated as a parameter and the node limit during testing is 10,000 nodes. A run was declared *successful* if testing was not halted due to repeating failures. The *learning cost* there is the total number of nodes during the learning phase of a run, calculated as the product of the average number of nodes per problem and the average number of problems per run. The *restart strategy* here is defined by a *full restart threshold (k, l)*, which performs a full restart after failure on $k$ problems out of the last $l$. (Here, $k = 3$ and $l = 4$.) This seeks to avoid full restarts when multiple but sporadic failures are actually due to uneven problem difficulty rather than to an inadequate weight profile. Problems that went unsolved under initial weights, before any learning occurred (*early failures*) were not counted toward full restart. If the first 30 problems went unsolved under the initial weights, learning was terminated. The learner's performance here is measured by the number of successful runs (out of 10) and the learning cost across a range of node limits.

Under every node limit tested, full restart produced more runs that were successful, as Figure 7 illustrates. At lower node limits, Figure 8 shows that better testing performance came with a learning cost similar to or slightly higher than the cost without full restart. At higher node limits, the learning cost was considerably lower with full restart. With very low node limits (200 or 300 nodes), even with full restart neither DWL nor RSWL was able to solve all the problems. During learning, many problems went unsolved under a low node limit and therefore did not provide training instances. On some (inadequate) runs under both methods, no solution was found to any of the first 30 problems, so learning was terminated without any weight changes. When the node limit was somewhat higher (400 nodes), more problems were solved, more training instances were available and more runs were successful. These reasonably low node limits set a high standard for the learner; only weight profiles well tuned to the class will solve problems within them and thereby provide good training instances. Further increases in the node limit (500, 600, 700 and 800 nodes), however, did not further increase the number of successful runs. Under higher node limits, problems were solved even with weight profiles that were not particularly good for the class, and may have produced training instances that were not appropriate. Under extremely high node limits (5000 nodes), problems were solved even under inadequate weight profiles, but the weight-learning mechanism was able to recover a good weight profile, and again the number of successful runs increased.

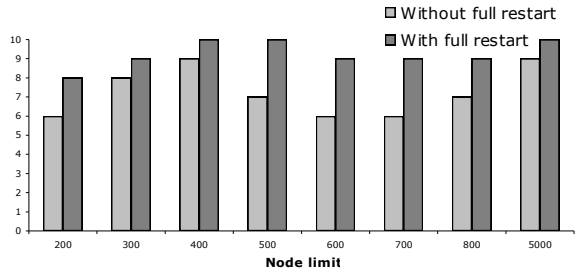



**Figure 7:** Number of successful runs (out of 10) on <30, 8, 0.26, 0.34> problems under different node limits.

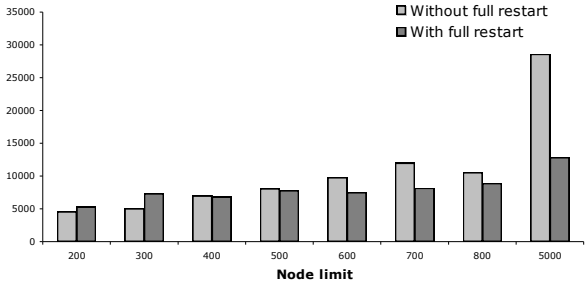



**Figure 8:** Learning cost, measured by the average number of nodes per run, on <30, 8, 0.26, 0.34> problems under different node limits.

Similar performance was observed under RSWL and on *Geo* and the other model B classes, but not on the *Comp* problems. Some *Comp* problems go unsolved under any node limit, while many others are solved. Because failures there are sporadic, they do not trigger full restart. The use of full restart on them does not improve learning, but it does not harm it either (data omitted). Full restart is therefore used throughout the remainder of our experiments.

### 7.2 Random subsets improve performance

Figure 9 shows the weights of six variable-ordering heuristics and their common benchmark after each of 30 problems. It illustrates weight convergence during learning with random subsets, and how some Advisors recovered from inadequate weights. Here the problems were drawn from <50, 10, 0.38, 0.2>, and 30% of the variable-ordering Advisors and 30% of the value-ordering 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, or the problem went unsolved, so that no learning or weight changes occurred. The first four problems were *early failures* (problems that went unsolved under initial weights, before any learning occurred). When the fifth problem was solved, some class-inappropriate Advisors received high weights from its training instances. On the next few problems, either highly-weighted but class-inappropriate heuristics were reselected and the problem went unsolved and no weights changed, or some class-appropriate Advisors were selected and gained high weights. Eventually the latter began to dominate decisions, so that the disagreeing class-inappropriate Advisors had their weights reduced. After the 21st problem, when the weight of *Min-static-connected-edges* significantly

decreased, the weights clearly separated the class-appropriate Advisors from the class-inappropriate ones. Afterwards, as learning progressed, the weights stabilized.

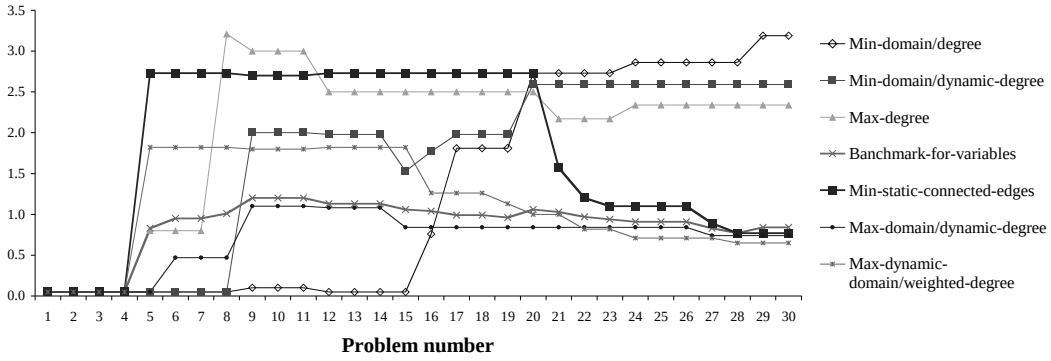



**Figure 9**: Weights of 7 Advisors during learning after each of 30 problems in <50, 10, 0.38, 0.2>, on a single run.

In experiments that illustrate the benefits of random subsets, we tested four different ways to choose the Advisors from each problem:

- All: Use all the Advisors on every problem.
- Fixed: Choose a fixed percentage $q$ of the variable-ordering Advisors and $q$ of the value-ordering Advisors, without replacement. Testing was performed for both $q$ = 30% and $q$ = 70%.
- Varying: For each problem, select a random percentage $r$ in [30, 70]. Choose $r$ percent of the variable-ordering Advisors and $r$ percent of the value-ordering Advisors, without replacement.
- Incremental: Initially select $q$ of the variable-ordering Advisors and $q$ of the value-ordering Advisors. Then, for each subsequent problem, increase the sizes of the random subsets in proportion to the number of Advisors whose weight is greater than their initially assigned weight.

On problems in <50, 10, 0.18, 0.37>, Table 4 compares each of these approaches for learning with random subsets of Advisors to learning with all the Advisors at once. When all 40 Advisors were consulted, the predominance of class-inappropriate Advisors sometimes prevented the solution of any problem under the given node limit, so that some learning phases were terminated after 30 unsolved problems. In those runs no learning occurred. With random subsets, however, adequate weights were learned on every run, and there were fewer early failures.

Table 4 also demonstrates that using random subsets significantly reduces learning time. The time to select a variable or a value is not necessarily directly proportional to the number of selected Advisors. This is primarily because dual pairs of Advisors share the same fundamental computational cost: calculating their common metric. For example, the bulk of the work for *Min-product-domain-value* lies in the one-step lookahead that calculates the products of the domain sizes of the neighbors after each potential value assignment. Consulting only *Min-product-domain-value* and not *Max-product-domain-value* will therefore not significantly reduce computational time. Moreover, the metrics for some Advisors are based upon metrics already

calculated for others. The reduction in total computation time per run is even more noteworthy, because it includes any reduction in the number of learning problems.

| *Advisors* | *Average number of early failures per run* | *Number of successful runs* | *Time per decision during learning* | *Learning time per run* |
|---|---|---|---|---|
| All | 27.0 | 4 | 100.00% | 100.00% |
| Fixed $q$ = 70% | 5.2 | 10 | 74.30% | 24.39% |
| Varying $r \in$ [30%,70%] | 1.9 | 10 | 65.84% | 20.74% |
| Incremental $q$ = 30% | 1.8 | 10 | 75.35% | 19.76% |
| Fixed $q$ = 30% | 3.1 | 10 | 55.39% | 21.85% |

**Table 4:** Early failures, successful runs, and percentage of computation time during learning with random subsets of Advisors, compared to computation time with all the Advisors on problems in <50, 10, 0.18, 0.37>.

The robustness of learning with random subsets is demonstrated with experiments in Table 5 that begin with fewer Advisors, a majority of which are class-inappropriate. Based on weights from successful runs with all Advisors, Advisors were first identified as class-appropriate or class-inappropriate for <50, 10, 0.18, 0.37> problems. ACE was then provided with two different sets $A_{var}$ of variable-ordering Advisors in which class-inappropriate Advisors outnumbered class-appropriate ones (9 to 6 or 9 to 4). When all the provided Advisors were consulted, the predominance of class-inappropriate Advisors effectively prevented the solution of any problem under the given node limit and no learning took place. When learning with random subsets, as the size of the random subsets decreased, the number of successful runs increased. As a result, random subsets with fixed $q$ = 30% is used throughout the remainder of these experiments.

| | **6 class-appropriate 9 class-inappropriate Advisors** | | **4 class-appropriate 9 class-inappropriate Advisors** | |
|---|---|---|---|---|
| *Advisors* | *Average number of early failures* | *Number of successful runs* | *Average number of early failures* | *Number of successful runs* |
| All | 30.0 | 0 | 30.0 | 0 |
| Fixed $q$ = 70% | 21.2 | 7 | 30.0 | 0 |
| Varying $r \in$ □ 30%, 70%] | 8.4 | 10 | 17.6 | 6 |
| Incremental $q$ = 30% | 3.8 | 10 | 12.0 | 9 |
| Fixed $q$ = 30% | 5.1 | 10 | 13.3 | 10 |

**Table 5:** Learning with more class-inappropriate than class-appropriate Advisors on problems in <50, 10, 0.18, 0.37>. Smaller, fixed-size random subsets appear to perform best.

### 7.3 The impact of decision difficulty

Table 6 illustrates that relative support and some assessment of problem difficulty make a difference. On problems in <50, 10, 0.38, 0.2>, RSWL solved more problems during both learning and testing than DWL, and required fewer full restarts. Moreover, both degree of difficulty variations of RSWL solved more problems in less space during testing.

| *Weight-learning algorithm* | *Learning* | | | *Testing* | |
|---|---|---|---|---|---|
| | *Problems* | *Unsolved problems* | *Full restarts* | *Nodes* | *Solved* |
| DWL | 36.8 | 14.1 | 0.8 | 13,708.58 | 91.8% |
| RSWL | 31.5 | 7.5 | 0.2 | 13,111.44 | **95.2%** |
| RSWL-*d*, *h*=30% | 30.9 | 8.4 | 0.1 | **11,849.00** | **94.6%** |
| RSWL-κ, *k*=0.2 | 32.9 | 8.9 | 0.3 | **11,231.60** | **95.0%** |

**Table 6:** Learning and testing performance with different preference expression methods on <50, 10, 0.38, 0.2> problems. Bold figures indicate statistically significant reductions, at the 95% confidence level, in the number of nodes and in the percentage of solved problems compared to search under DWL.

### 7.4 Performance with preferences

We tested both linear interpolation and the Borda methods on all problem classes described in Section 5. On the unstructured model B problems, preference expression made no significant difference. On *Comp*, however, across a broad range of node limits, preference expression had an effect, as shown in Table 7. Going beyond ordinary Borda methods, we also emphasized (*enhanced*) the influence of large sets of choices that share the same score. The enhanced methods make the size of a point inversely proportional to the number of subsets of tied values, and assign strengths to every commented choice instead of only to those with the favored *f* scores. Initially, most variables in the central component score similarly on most metrics, and most variables in the satellite score similarly to one another but differently from those in the central component. Under the enhanced version of *Borda–wt*, 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, in the enhanced version the next lower subset will have a much lower strength, which decreases its influence. Moreover, when many choices share the same score, they are penalized for failure to discriminate, and their strength is lowered. When *enhanced Borda–w* assigns lower strengths to large subsets from the central component, it makes them less attractive. That encourages the variables from subproblem defined by the satellite to be selected first; this is often the right way to solve such problems.

| *Node limit* | *Ranking* | | *Borda-w* | | *Borda-wt* | | *Borda-w enhanced* | | *Borda-wt enhanced* | | *Linear* | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *Nodes* | *Solved* | *Nodes* | *Solved* | *Nodes* | *Solved* | *Nodes* | *Solved* | *Nodes* | *Solved* | *Nodes* | *Solved* |
| 5000 | 161.1 | 97.8% | 139.9 | 98.0% | 151.1 | 98.0% | 134.1 | 98.0% | 638.5 | 88.6% | 164.2 | 97.80% |
| 1000 | 161.1 | 97.8% | 130.1 | 98.2% | 183.8 | 97.4% | 134.1 | 98.0% | 564.7 | 89.8% | 164.2 | 97.80% |
| 500 | 161.5 | 97.8% | 130.5 | 98.2% | 150.7 | 98.0% | 121.1 | 98.2% | 728.5 | 86.6% | 164.2 | 97.80% |
| 100 | 161.4 | 97.8% | 139.4 | 98.0% | 190.0 | 97.4% | 111.6 | 98.4% | 642.1 | 88.2% | 163.7 | 97.80% |
| 35 | 160.4 | 97.8% | 147.6 | 98.0% | 128.5 | 98.4% | 111.7 | 98.4% | 660.0 | 89.0% | **33.5** | 100.0% |

**Table 7:** Testing performance with RSWL on *Comp* problems with reduced node limits and a variety of preference expression methods. Although there appear to be substantial differences, the variance is such that only the figure in bold is a statistically significant improvement at the 95% confidence level.

Linear interpolation performed similarly to RSWL with ranking on *Comp* problems, except under the lowest node limit tested. Given only 35 nodes, RSWL with linear preference expression was able to solve every problem during testing. The 35-node limit imposes a very high learning standard; it allows learning only from highly space-efficient solutions. (A backtrack-free solution would expand exactly 30 nodes for a *Comp* problem.) Only with the nuances of information provided by linear preference expression did ACE develop a weight profile that solved all the testing problems in every run.

## 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, one 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 these ideas on several fronts. Under an option called *Pusher*, ACE consults the single highest-weighted tier-3 variable-ordering heuristic below the maximum search depth at which it has experienced backtracking on other problems in the same class (Epstein, et al., 2005). Current work includes learning different weight profiles for different stages in solving a problem, where stages are determined by search tree depth or the constrainedness of the subproblem at the decision point. A generalization of that approach would associate weight profile(s) with an entire benchmark family of problems, and begin with the weights of the most similar benchmark family for each new problem instance.

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 (Schapire, 1990). A major focus is the automated selection of good parameter settings for an individual class (including the node limit and full-restart parameters), given the results in (Hutter, et al., 2006). 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 (Wallace, 2005). Meanwhile, ACE does a good job of tailoring a mixture of search heuristics to each new problem class it encounters.

## 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 (Smith and Grant, 1998), 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, (Boussemart, et al., 2004), and ratio of dynamic domain size to weighted degree. 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.

## Acknowledgements

ACE is a joint project with Eugene Freuder and Richard Wallace of The Cork Constraint Computation Centre. Dr. Wallace made important contributions to the DWL algorithm. This work was supported in part by the National Science Foundation under IRI-9703475, IIS-0328743, and IIS-0739122.

## References

K. I. Aardal, S. P. M. v. Hoesel, A. M. C. A. Koster, C. Mannino and A. Sassano. Models and solution techniques for frequency assignment problems. *4OR: A Quarterly Journal of Operations Research* 1(4):261-317, 2003.

K. Ali and M. Pazzani. Error reduction through learning multiple descriptions. *Machine Learning* 24:173-202, 1996.

J. E. Borrett, E. P. K. Tsang and N. R. Walsh. Adaptive Constraint Satisfaction: The Quickest First Principle. In *Proceedings of the Twelfth European Conference on Artificial Intelligence (ECAI-96)*, pages 160-164, Budapest, Hungary, 1996.

F. Boussemart, F. Hemery, C. Lecoutre and L. Sais. Boosting Systematic Search by Weighting Constraints. In *Proceedings of the Sixteenth European Conference on Artificial Intelligence (ECAI-04)*, pages 146-150, Valencia, Spain, 2004.

T. G. Dietterich. Ensemble methods in machine learning. In *Proceedings of the First International Workshop on Multiple Classifier Systems*, pages 1-15, Cagliari, Italy, 2000.

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

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

Y. Freund and R. Schapire. Experiments with a new boosting algorithm. In *Proceedings of the Thirteenth International Conference on Machine Learning (ICML-96)*, pages 148-156, Bari, Italy, 1996.

A. S. Fukunaga. Automated Discovery of Composite SAT Variable-Selection Heuristics. In *Proceedings of the AAAI/IAAI-02*, pages 641-648, Menlo Park, CA, 2002.

M. Gagliolo and J. Schmidhuber. Dynamic Algorithm Portfolios. In *Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, FL, 2006.

I. P. Gent, P. Prosser and T. Walsh. The Constrainedness of Search. In *Proceedings of AAAI/IAAI* -96, pages 246-252, Portland, OR, 1996.

C. Gomes, C. Fernandez, B. Selman and C. Bessière. Statistical Regimes Across Constrainedness Regions. In *Proceedings of the Tenth Conference on Principles and Practice of Constraint Programming (CP-04)*, pages 32-46, Toronto, Canada, 2004.

C. P. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence* 126:43-62, 2001.

L. Hansen and P. Salamon. Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 12:993-1001, 1990.

T. Hulubei and B. O'Sullivan. Search Heuristics and Heavy-Tailed Behavior. In *Proceedings of the Principles and Practice of Constraint Programming (CP-05)*, pages 328-342, Sitges, Spain, 2005.

F. Hutter, Y. Hamadi, H. H. Hoos and K. Leyton-Brown. Performance Prediction and Automated Tuning of Randomized and Parametric Algorithms. In *Proceedings of the Principles and Practice of Constraint Programming (CP-06)*, pages 213-228, Nantes, France, 2006.

D. S. Johnson, C. R. Aragon, L. A. McGeoch and C. Schevon. Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning. *Operations Research* 37:865-892, 1989.

J. Kivinen and M. K. Warmuth. Averaging expert predictions. In *Proceedings of the Computational Learning Theory: 4th European Conference (EuroCOLT '99)*, pages 153-167, Nordkirchen, Germany, 1999.

C. Lecoutre, F. Boussemart and F. Hemery. Backjump-based techniques versus conflict directed heuristics. In *Proceedings of International Conference on Tools with Artificial Intelligence (ICTAI-04)*, pages 549–557, Boca Raton, FL, 2004.

S. Minton, J. A. Allen, S. Wolfe and A. Philpot. An Overview of Learning in the Multi-TAC System. In *Proceedings of the First International Joint Workshop on Artificial Intelligence and Operations Research*, Timberline, OR, 1995.

A. Nareyek. Choosing Search Heuristics by Non-Stationary Reinforcement Learning. In *Metaheuristics: Computer Decision-Making*, 523-544, Kluwer Academic Publishers, 2004.

D. Opitz and J. Shavlik. Generating accurate and diverse members of a neural-network ensemble. *Advances in Neural Information Processing Systems* 8:535-541, 1996.

L. Otten, M. Grönkvist and D. P. Dubhashi. Randomization in Constraint Programming for Airline Planning. In *Proceedings of the Principles and Practice of Constraint Programming (CP-06)*, pages 406-420, Nantes, France, 2006.

K. E. Petrie and B. M. Smith. Symmetry breaking in graceful graphs. In *Proceedings of the Principles and Practice of Constraint Programming (CP-03)*, pages 930-934, Kinsale, Ireland, 2003.

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

S. V. Petrovic and S. L. Epstein. Relative Support Weight Learning for Constraint Solving. In *Proceedings of the Workshop on Learning for Search at AAAI-06*, pages 115-122, Boston, MA, 2006.

S. V. Petrovic and S. L. Epstein. Preferences Improve Learning to Solve Constraint Problems. In *Proceedings of the Workshop on Preference Handling for Artificial Intelligence at AAAI-07*, pages 71-78, Vancouver, Canada, 2007.

S. V. Petrovic and S. L. Epstein. Random Subsets Support Learning a Mixture of Heuristics. *International Journal on Artificial Intelligence Tools (IJAIT)* 17:501-520, 2008.

G. Ringwelski and Y. Hamadi. Boosting Distributed Constraint Satisfaction. In *Proceedings of the Principles and Practice of Constraint Programming (CP-05)*, pages 549-562, Sitges, Spain, 2005.

Y. Ruan, H. Kautz and E. Horvitz. The backdoor key: A path to understanding problem hardness. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, pages 124-130, San Jose, CA, 2004.

D. G. Saari. *Geometry of Voting*, Springer-Verlag, Berlin, 1994.

D. Sabin and E. C. Freuder. Understanding and Improving the MAC Algorithm. In *Proceedings of Principles and Practice of Constraint Programming (CP-97)*, pages 167-181, Schloss Hagenberg, Austria, 1997.

R. E. Schapire. The strength of weak learnability. *Machine Learning* 5(2):197-227, 1990.

B. Smith and S. Grant. Trying Harder to Fail First. In *Proceedings of the European Conference on Artificial Intelligence (ECAI-98)*, pages 249-253, Brighton, UK, 1998.

M. Streeter, D. Golovin and S. F. Smith. Combining multiple heuristics online. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07)*, pages 1197–1203, Vancouver, Canada, 2007.

G. Valentini and F. Masulli. Ensembles of learning machines. In *Proceedings of the Neural Nets WIRN Vietri-02*, pages 3-19, Vietri sul Mare, Italy, 2002.

R. J. Wallace. Factor analytic studies of CSP heuristics. In *Proceedings of the Principles and Practice of Constraint Programming (CP-05)*, pages 712-726, Sitges, Spain, 2005.

H. P. Young. Condorcet's Theory of Voting. *The American Political Science Review* 82:1231-1244, 1988.