Relative Support Weight Learning for Constraint Solving

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

[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
spetrovic@gc.cuny.edu, susan.epstein@hunter.cuny.edu

Abstract

Many real-world problems can be represented and solved as constraint satisfaction problems, but their solution requires the development of effective, efficient constraint solvers. A constraint solver's success depends greatly upon the heuristics chosen to guide search. Some heuristics perform well on one class of problems, but are less successful on others. The solver described here learns a weighted combination of heuristic recommendations customized for a given class of problems. A pre-existing algorithm has weights learned for heuristics from the number of nodes explored during learning. We present here a new algorithm which learns weights for heuristics that consider the nuances of relative support for actions. The resultant performance is statistically significantly better than that of traditional individual heuristics.

Introduction

This work uses subproblem difficulty and the nuances available in the expression of heuristics preferences to guide search and learning effectively. In the domain investigated here, a semi-supervised learner searches for solutions to difficult combinatorial problems. The learner gleans training instances from its own (likely imperfect) trace, and uses them to refine its search algorithm before it continues to the next problem. Our long-term goal is to learn an effective, efficient combination of domain-specific heuristics that works well on a specific class of problems. This paper moves toward that goal with a new algorithm that learns weights to balance search heuristics.

The program described here learns to solve constraint satisfaction problems by refining the weights assigned to a large set of heuristics. It generates training instances from its own search experience. The program's decisions are guided by a weighted combination of traditional constraint-solving heuristics. The learning algorithm accepts recommendations that assign some degree of support to each available choice, compares them to the performed action, and updates the heuristics' weights accordingly. Search and learning are closely coupled; weights are updated based on search results, and decisions made during search are based on the current weights.

DWL (Digression-based Weight Learning) is a pre-existing weight-learning algorithm for such a task. Its weight reinforcements are based upon the size of the search tree relative to searches on previous problems, and to the sizes of *digressions* (failed subtrees) (Epstein, Freuder, et al., 2005). We introduce here *RSWL* (Relative Support Weight Learning), an algorithm for the same task. RSWL bases its weight reinforcements on the degree of support from a heuristic to all the available choices, the difficulty of the current subproblem, whether the decision is early or late in the search, the number of choices available, and the weights at the moment of decision.

The next two sections of this paper provide background on combining heuristics and on constraint satisfaction. Subsequent sections describe the learning architecture used here, and how the two weight-learning algorithms work within it to combine heuristics. Later sections detail the experiments and discuss the results.

Combining heuristics

Ensemble learning methods select hypotheses from a hypothesis space and combine their predictions (Russell and Norvig, 2003). A hypothesis can be represented by a function, as the output of an algorithm, or as the opinion of an expert or a heuristic. Learning to combine experts has been theoretically analyzed for supervised environments (Kivinen and Warmuth, 1999). A *mixture of experts algorithm* learns from a sequence of trials how to combine experts' predictions. A trial consists of 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, all experts' predictions and observed values are created to maximize the relative loss. The best expert is not known in advance, and finding the best expert in an *on-line setting* (where there is no separate set of training instances and a decision must be produced at any time based on previously experienced instances) is impossible. Mixture of experts algorithms have been proved asymptotically close to the behavior of the best expert (Kivinen and Warmuth, 1999).

Our setting differs from a traditional mixture of experts in several critical ways. In solving constraint satisfaction problems, the problem class is given in advance, and all the experts' predictions are based upon the characteristics of the problem. Thus, a program that learns to solve constraint satisfaction problems with a mixture of experts does not seek to improve the worst-case performance, but to customize the mixture of experts for the given class. In such a setting, a mixture of experts can often outperform any single expert (Valentini and Masulli, 2002). Our learner works in a semi-supervised environment, without full information about the correctness of decisions. Finally, it learns in an off-line setting, where the goal is to accomplish some level of competence on testing problems.

There are several reasons why a combination of experts can offer improved performance, compared to a single expert (Valentini and Masulli, 2002). First, there may be no single expert that is best on the all problems. A combination of experts could thus enhance the accuracy and reliability of the overall system. Next, it may be possible to use a natural decomposition of the problem, learn only from relevant instances, or monitor different features during learning. In addition, on limited data, there may be different hypotheses that appear equally accurate. In this case, one could approximate the unknown true hypothesis by the simplest one, but averaging or mixing all of them gives a better approximation. A combination of hypotheses can expand the space of representable functions. The target function may not be representable by any hypothesis in the pool, but their combination could produce an acceptable representation. Finally, the Condorcet Jury Theorem, proposed by the Marquis de Condorcet (1745-1794), considers the conditions under which a democracy as a whole is more effective than any of its constituent members. The theorem states that the judgment of a committee of competent jurors (each of which is correct with some probability greater than 0.5) is superior to the judgment of any individual juror.

Constraint satisfaction problems

A *constraint satisfaction problem* (CSP) consists of a set of variables with associated domains and a set of constraints, expressed as relations over subsets of those variables. A *partial instantiation* of a CSP is an assignment of values to some proper subset of the variables. An instantiation is *consistent* if and only if all constraints over the variables in it are satisfied. A *solution* to a CSP is a consistent instantiation of all its variables. Determining whether a CSP has a solution is an NP-complete problem; the worst-case cost is exponential in the number of variables for any known algorithm. Most instances of these problems can be solved with a cost much smaller than the worst-case cost, however. Problems that may be expressed as CSPs include scheduling, satisfiability (SAT), and graph coloring. A *binary CSP* (all constraints are on at most two variables) 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.

A *class* is a set of CSPs with the same characterization. For example, CSPs may be characterized by the parameters $<n, m, d, t>$, where n is the number of variables, m the maximum domain size, d the density (fraction of edges out of $n(n\text{-}1)/2$ possible edges) and t the *tightness* (fraction of possible value pairs that each constraint excludes) (Gomes, Fernandez, et al., 2004). A class can also restrict the structure of the problem. For example, a *composed problem* consists of a subgraph called its *central component* joined to one or more subgraphs called satellites. (These were in part inspired by the hard manufactured problems of (Bayardo and Schrag, 1996).) The number of variables n, maximum domain size m, density d, and tightness t of the central component and its satellites are specified separately, as are the density and tightness of the links between them. A class of composed graphs is specified here as

$$<n, m, d, t>\ s\ <n, m, d, t>$$

where the first tuple describes the central component, s is the number of satellites, and the second tuple describes the satellites. (Satellites are not connected to one another.) Although CSPs in the same class are ostensibly similar, there is evidence that their difficultly may vary substantially for a given solution method (Hulubei and O'Sullivan, 2005). Our experiments here are limited to classes of solvable binary CSPs, generated randomly using methods and code shared by the CSP community.

The CSP search algorithm used here alternately selects a variable and then selects a value for it from its domain. Thus it incrementally extends a partial instantiation by a value assignment consistent with all previously-assigned values. When an inconsistency arises, search *backtracks* chronologically: the subtree rooted at the inconsistent node is pruned and another value from the domain of the same variable is tried. If every value in that variable's domain is inconsistent, the current partial instantiation cannot be extended to a solution, so the previously assigned variable in the instantiation is reassigned. Typically, extensive search occurs when many attempts lead to "near" solutions, and backtracking occurs deep in the search tree. Further pruning of the search tree is accomplished by some form of *propagation* to detect values that cannot be supported by the current instantiation. Here we use *MAC-3* to maintain arc consistency during search (Sabin and Freuder, 1994). MAC-3 temporarily removes currently unsupportable values to calculate *dynamic domains* that reflect the current instantiation. The size of the search tree depends upon the order in which values and variables are selected. Different *variable-ordering heuristics* (rules to select the next variable for instantiation) and *value-ordering heuristics* (rules to select the next value to be assigned to an already-selected variable) can support extensive early pruning and thereby speed search.

Solving with heuristics

ACE (the Adaptive Constraint Engine) learns to customize a weighted mixture of heuristics for a given CSP class (Epstein, et al., 2005). ACE is based on FORR, an architecture for the development of expertise from multiple heuristics (Epstein, 1994). ACE makes decisions by combining recommendations from procedures called *Advisors*, each of which implements a heuristic for taking, or not taking, an action. By solving instances of problems from a given class, ACE learns an approach tailored to that class. Advisors are organized into three tiers. Tier-1 Advisors are always correct. If a tier-1 Advisor comments positively, the action is executed; if it comments negatively, the action is eliminated from further consideration during that decision. The only tier-1 Advisor in use here is *Victory*, which recommends any value from the domain of the last unassigned variable. Tier-1 Advisors are consulted in a user-specified order. Tier-2 Advisors address subgoals; they are outside the scope of this paper. The decision making described here focuses on the heuristic Advisors in tier 3.

Each tier-3 Advisor can *comment* upon any number of choices, and each comment has a *strength* which indicates the degree of support from the Advisor for the choice. Each Advisor's heuristic view is based upon a descriptive metric. An example of a value-selection Advisor is *Max Product Domain Value*, which recommends values that maximize the product of the sizes of the dynamic domains of its neighbors. One example of a variable-selection Advisor is *Min dom/deg*, which recommends variables whose ratio of dynamic domain size to static degree in the constraint graph is a minimum. Another example is *Min dom/ddeg* that minimize the ratio of dynamic domain size to dynamic degree. For each metric, there are two Advisors, one that favors smaller values for the property and one that favors larger values. For example, domain size is referenced by *Max Domain* and *Min Domain,* which recommend variables with the largest and smallest initial domains, respectively. Typically, one Advisor from each pair is known by CSP researchers to be a good heuristic, but ACE implements both of them, and occasionally demonstrates that the less accepted heuristic is successful for some problem class. There are also two *benchmark Advisors*, one for value selection and one for variable selection. They model random tier-3 advice but do not participate in decisions. During testing, only an Advisor that has a weight larger than the weight of its respective benchmark is permitted to comment. When a decision is passed to tier 3, all its Advisors are consulted together, and a selection is made by *voting:* the action with the greatest sum of weighted strengths from all comments is executed.

Learning weights

ACE uses a weight-learning algorithm to update its *weight profile,* the set of weights of its tier-3 Advisors. *Positive training instances* are those made along an error-free path extracted from a solution. *Negative training instances* are value selections at the root of a digression and variable selections when a value assignment to the selected variable fails. Decisions made within a digression are not considered.

The only information available to ACE comes from its limited experience as it finds one solution to a problem. This approach is problematic for several reasons. There is no guarantee that some other solution could not be found much faster, if even a single decision were different. Although variable selections are always correct (because with correct value assignments, any variable ordering leads to a backtrack-free solution), there can be a large difference in search performance for different variable orderings. Without supervision, we must somehow declare what constitutes a correct variable selection. Here, a variable selection is considered correct if no value assignment to that variable subsequently failed. All we know for certain is that the value selection at the root of each digression tree is wrong, based upon the evaluation of an entire subtree. Yet, that failure may be the result of some earlier decision. Moreover, a particular Advisor may be incorrect on some decisions that resulted in a large digression, and still be correct on many other decisions in the same problem. These issues are addressed with two different weight learning algorithms: DWL and RSWL.

DWL: Digression-based Weight Learning

DWL determines that an Advisor supports a decision if it is included among the Advisor's highest-strength preferences (Epstein, et al., 2005). Advisors that support a positive training instance are 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. Advisors that support a negative training instance are penalized in proportion to the number of search nodes in the resultant digression. Short solutions indicate a good variable order, so correct variable-ordering Advisors on training instances from a successful short search are highly rewarded. For value-ordering Advisors, short solutions 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 increments for correct Advisors. A successful long search, in contrast, suggests that a problem was relatively difficult, so value-ordering Advisors that supported positive training instances there receive substantial weight increments.

RSWL: Relative Support Weight Learning

Recall that, in ACE, an Advisor does not select one variable or value, but recommends multiple choices each with some numeric strength, the Advisor's degree of support (Epstein and Freuder, 2001). Although ACE incorporates those strengths in voting, DWL attends only to which assigned strength is the highest. For example, if one Advisor recommends the correct choice but only with its second-highest strength and another does not recommend it at all, both Advisors will be equally penalized.

In contrast, RSWL considers all recommendation strengths, not only the highest. 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. Under RSWL an Advisor is deemed to support an action if its relative support for that action is positive, and to oppose it if its relative support is negative. Reinforcement (positive or negative) is proportional to relative support. Another crucial difference between DWL and RSWL is that DWL bases rewards on the size of the search tree, while RSWL bases them on properties of the state in which a decision is made.

For weight reinforcement, RSWL credits Advisors that support correct actions and penalizes those that support incorrect actions. In the experiments recounted in this paper, we explored RSWL with various combinations of the following factors:

• *Relative support*. Support for a correct decision with near-average strength earns only a small credit, while support with substantially greater than average strength indicates a clear preference and receives a proportionally larger credit.

• *Constrainedness*. κ is a parameter that can be used to identify hard classes of problems (Gent, Prosser, et al., 1999). For CSPs, κ depends upon *n, d, m,* and *t* (defined above):

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

Regardless of the algorithm used, for fixed *n* and *m*, hard problem classes have κ close to one. Although κ was intended for a class, RSWL uses it as a measure of subproblem difficulty throughout search. RSWL rewards an Advisor only when it supports a correct decision on a training instance derived from a search state where κ is greater than some specified value. Our rationale for this is that, on easy problems, any decision leads to a solution. Credit for an easy decision would effectively increase the weights of Advisors that led to it, whose weights were presumably already high. Similarly, RSWL penalizes an Advisor only when it supports an incorrect decision and the corresponding state has κ lower than specified value, because there should be no errors on easy problems.

• *Depth of the search tree*. Because calculating κ on every training instance is computationally expensive, we also investigated RSWL with the depth of the search tree as a rough, quick estimate of problem hardness. Early decisions (at the top of the search tree) are known to be more difficult (Ruan, Kautz, et al., 2004).

• *Number of available choices*. Another factor in learning is the number of choices presented to the Advisors. For variable-selecting Advisors, the number of available choices is *n* minus the current search tree depth (number of assigned variables). For value-selecting Advisors, the number of available choices is the domain size of the currently-selected variable, and varies due to propagation.

• *Current weight*. Advisors with high weights should be reliable and accurate. When a highly-weighted Advisor supports an incorrect decision, RSWL imposes an additional penalty. Similarly, when an Advisor with a low weight supports a correct decision, additional credit is given, so that its weight can recover faster.

Weight convergence

As calculated by both DWL and RSWL, the weight of an Advisor is directly proportional to the rewards and penalties it receives, but inversely proportional to the number of training instances on which it comments. As a result, each weight eventually achieves *stability* (its standard deviation over some number of consecutive problems becomes very small). In most situations, learned weights elicit good performance. Occasionally, however, good heuristics are unable to control decision making and *anti-heuristics* (heuristics that make poor choices) are repeatedly rewarded. DWL takes a radical approach to this problem: a full restart of the learning process (Petrovic and Epstein, 2006). DWL recognizes when its current learning attempt is not promising, abandons the responsible training problems, and restarts the entire learning process.

In contrast, RSWL uses a recovery mechanism motivated by the share update algorithm (Herbster and Warmuth, 1998). When ACE is not able to solve a problem within the specified *step limit* (maximum number of search decisions), RSWL redistributes (*shares*) the weights of highly-weighted Advisors among all the Advisors, so that small weights can recover quickly. This reapportionment spreads equally among all Advisors a (parameterized) percentage of the weights of Advisors whose weights are greater than their benchmarks' weights. Such sharing can be either across all Advisors or by category, so that variable-selection Advisors share among themselves, and value-selection Advisors share among themselves, possibly with different a reapportionment percentage.

Experimental design

DWL, RSWL, and three traditional heuristics (min domain, *dom/deg,* and *dom/ddeg*, described earlier) were tested on random, solvable, binary CSPs from 4 classes: <30, 8, 0.16, 0.5>, <30, 8, 0.18, 0.5>, the *general* class <30, 8, 0.31, 0.34> with κ near 1 (indicating that these problems are particularly difficult for their *n* and *m* values), and the *composed* class

<22, 6, 0.6, 0.1> <8, 6, 0.72, 0.45>.

These composed problems have 30 variables, each with domain size 6, separated into a central component of 22 variables that is relatively loose and a satellite of 8 variables that is somewhat more dense and significantly tighter. Links between the central component and a satellite occur with density 0.115 and tightness 0.05. Composed problems present a particular challenge to most traditional heuristics, as shown in the next section.

All experiments were conducted within 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* in ACE is a sequence of fresh problems to be solved with learning turned off. For each problem class and for every algorithm, each testing phase used the same 20 problems. A *run* in ACE is a learning phase followed by a testing phase, and draws all its problems from a single class. No learning phase was used with the traditional heuristics.

In the work reported here, during learning ACE referenced the 36 tier-3 Advisors described in the appendix. During testing, ACE included only those tier-3 Advisors whose weights exceeded their corresponding benchmarks.

Table 1: A blend of heuristics outperforms individual ones. Performance of three traditional (non-learning) heuristics and the DWL algorithm on problems from the class <30,⊠,□.31,□.34>.

Approach	**Learning tasks**	**Testing steps**
min domain	—	335.20
dom/deg	—	219.70
dom/ddeg	—	201.40
DWL	16.90	139.75

Learning stopped when ACE solved 10 consecutive problems (the *expertise* criterion).

During learning, step limits on individual problems must be set high enough to allow ACE to find solutions, but not so high that search wanders and thereby produces poor training examples and learns poor weight profiles. Step limits were 2000 for the general CSPs and 5000 for composed CSPs. During testing, step limits were 20000 for general CSPs and 5000 for composed CSPs. During learning, DWL used full restart when 4 out of the last 7 consecutive tasks failed. Unless otherwise specified, RSWL only gave credit for the support of correct decisions (i.e., no penalties). Without sharing, RSWL learns to solve the general problems on some runs, but fails on others. Therefore we established the following sharing policy as the default: share 40% of the weight of all Advisors whose weight is higher than its benchmark Advisor's weight.

Learning performance for DWL and RSWL was measured by the average number of learning problems over 10 runs. Testing performance for all algorithms was the number of steps per problem, averaged over 10 runs.

Results

DWL, RSWL, and all the traditional heuristics performed uniformly well on the <30,⊠,□.16,□.5> and <30,⊠,□.18,□.5> classes. (Those results are omitted here.) We demonstrate first the power of a combination of heuristics. Table 1 shows the performance of some of often used traditional heuristics and of the pre-existing DWL learning algorithm with 36 heuristics on the general problems. "Minimize the dynamic domain size", "minimize the ratio of dynamic domain size to static degree" (*dom/deg*), and "minimize the ratio of dynamic domain size to dynamic degree" (*dom/ddeg*) are all traditional heuristics that work well on a wide range of problems. The blend of heuristics that DWL creates is significantly better than each of them. The remainder of our empirical work studies RSWL.

Table 2: Performance varies with bounds for □ and for search tree depth. Data is for RSWL on problems from the class <30,⊠,□.31,□.34>. Only credit was assigned, with sharing at 40% among all Advisors.

Criterion	**Learning tasks**	**Testing steps**
□ > 0.8	21.60	174.91
□ > 0.6	13.30	143.33
□ > 0.4	19.40	152.24
Depth < 5	14.20	137.21
Depth < 10	12.80	140.13
Depth < 15	12.90	143.21
Depth < 20	12.70	156.71

On the general problems

First we look at criteria for reinforcement, and then at the degree of reinforcement, special consideration for low-weight Advisors that comment correctly and high-weight Advisors that comment incorrectly, and sharing.

Table 2 illustrates the importance of constrainedness and the depth of the search tree to weight learning with RSWL on the problems from the general class. Credit was given to Advisors that supported a correct decision, either when the problem was sufficiently difficult (□) or when search was high enough in the tree (depth). When credit was given only on extremely hard subproblems (□ > 0.8), there were too few learning instances to refine the weights. Credit that extended to relatively easy subproblems (□ > 0.4) proved unjustified, since the supported decisions were not necessarily better than the other available choices. Testing problems were solved in fewer steps and learning required fewer tasks when □⊠□.6. Search tree depth as a criterion for credit did best when credit was limited to early decisions. Although search tree depth as a criterion seems to show slightly better performance, the choice of a lower bound for □ is more likely to be independent of the problem class.

Thus far we have used □ and search tree depth as criteria for whether or not reinforcement should be given. Another approach is to use parameters to determine how much reinforcement to apply. For example, instead of credit = *rs* for correct decisions when □ > 0.6, we might credit every correct decision by

$$credit = 1.7 \cdot \square \cdot rs \qquad [1]$$

where *rs* is an Advisor's relative support. The use of 1.7 in

Table 3: Performance varies with emphasis on relative support. Data is for RSWL on problems from the class <30,⊠,□.31,□.34>. Credits and penalties were a multiple of relative support, as defined in the text. Sharing was at 40% among all Advisors.

Reinforcement computation	**Learning tasks**	**Testing steps**
For credit: 1.7 ·□	14.70	141.81
For credit: the number of choices	12.10	135.50
For credit: the number of choices For penalty: the complement of the number of choices	11.50	138.10

Table 4: Performance changes when additional credit is given to low-weight, correct Advisors or high weight incorrect Advisors, as described in the text. Data is for RSWL on problems from the class <30,□8,□0.31,□0.34>. Some data is repeated here to facilitate comparison with earlier tables. * indicates that learning failed. Sharing was at 40% among all Advisors.

Weight factor	Criterion	Learning tasks	Testing steps
—	□ > 0.6	13.30	143.33
2	□ > 0.6	13.70	142.11
3	□ > 0.6	14.50	132.23
5	□ > 0.6	—	*
—	Depth < 5	14.20	137.21
2	Depth < 5	19.10	133.82
—	Depth < 10	12.80	140.13
2	Depth < 10	11.50	132.09
—	Credit: □ > 0.6 Penalty: □ < 0.1	14.80	147.12
Credit: 2 Penalty: 1	Credit: □ > 0.6 Penalty: □ < 0.1	13.20	136.35

[1] provides credit ≈ rs when k ≈ 0.6, slightly more credit with □□>□0.6, and some reduced credit when □ < 0.6 (instead of none at all). The results appear in the first line of Table 3. For the remainder of Table 3, we calculated credit as the product of relative support and number of available choices, and penalties as the product of relative support and the difference between the number of initial choices and the number of available choices. (This increases penalty size when an incorrect decision is made from fewer choices.)

Table 5: Performance changes with different sharing approaches. Data is for RSWL on problems from the class <30,□8,□0.31,□0.34>. Credit was given only when depth□<□10, and no penalties were assigned. * denotes that without sharing, learning failed.

Sharers	%	Learning tasks	Testing steps
None	—	*	—
All	40%	12.80	140.13
All	25%	15.80	130.62
Variable Advisors Value Advisors	40% 25%	11.60	135.84
Variable Advisors Value Advisors	25% 25%	14.50	143.50
Variable Advisors Value Advisors	40% 40%	11.60	133.46

Table 4 shows the improvement in performance when extra credit is given to low-weight Advisors that support the correct decision. In these experiments, an Advisor with weight *w* received credit that was the product of its relative support with respect to a weight factor *wf* as follows:

$$credit = \square \begin{array}{ll} \square rs & \text{when } w > wf \,\square 1 \\ \square rs \cdot (wf \,\square\, w) & \text{otherwise} \end{array} \qquad [2]$$

When the weight factor is associated with a penalty this becomes:

$$credit = \square \begin{array}{ll} \square rs & \text{when } w < 1 \,\square\, wf \\ \square rs \cdot (wf + w) & \text{otherwise} \end{array} \qquad [3]$$

In experiments with the weight factor, the average number of learning tasks may increase slightly. When learning is progressing well, this feature does not significantly affect weights, because low-weight Advisors are usually anti-heuristics. In the case of repetitive errors, which indicate the need for a more radical change in the weight profile, the weight factor helps the weights of good Advisors recover faster. Clearly, an overly high weight factor can prevent learning. With a weight factor of 5, learning never converged and was terminated after 100 learning tasks.

The data in Table 5 shows how sharing, as implemented here, can recover when a run is not progressing well. Table 5 indicates that appropriate values for reapportionment parameters can support improved performance.

On the composed problems

Table 6 illustrates performance on the composed problems described earlier. If an early variable choice is from the satellite, the problem will be solved quickly. On the other hand, if an early variable choice is from the central component, propagation will impact domains primarily in the central component and solve it, but be unable to extend that solution to the satellite. Such searches are typically extremely long. We used a 5000-step limit here, but in other experiments with a 20000-step limit, many problems in this

Table 6: RSWL outperforms other algorithms on 20 composed problems from <22,□6,□0.6,□0.1>□□<8,□6,□0.72,□0.45>. Sharing was at 40% among all Advisors; credits and penalties assigned as noted.

Approach	Testing steps	Solved testing problems
min domain	308.40	95.0%
dom/deg	551.90	90.0%
dom/ddeg	306.65	95.0%
DWL	549.82	91.5%
RSWL credit when depth < 15, penalty when depth > 15	137.67	98.5%
RSWL credit when depth < 10	86.90	99.5%

class still went unsolved by the traditional heuristics, with considerably more testing steps. For this reason Table 6 includes the number of problems solved during testing as another measure of performance.

It is also important to note that on composed problems, ACE often learns so-called anti-heuristics to be the best choices. For example, in one successful run (where 11 of the 20 testing problems were without any retraction at all), RSWL learned a weight of 0.15 for the well-known heuristic max degree, while its opposite, min degree, had a weight of 10.71.

Discussion

The experiments recounted here show that a variety of factors impact RSWL's learning and ultimately its search performance: the constrainedness of the current subproblem, the depth of the search tree, the number of available choices, the weight factor, the percentage of shared weights and whether or not penalties are enforced. With one or more of these in place, RSWL performs well on selected classes of CSP problems. With well-chosen parameters, on all the problem classes tested here, RSWL performs significantly better than traditional heuristics and comparably to a preexisting algorithm that was based upon different features. RSWL's mechanism for faster weight recovery speeds convergence to good weights.

This research will be extended in several directions. We intend to study RSWL on other classes of CSPs, including quasigroups with holes, small world problems, and unsolvable general problems. We plan to determine whether ideal values for parameters are problem-class dependent. If so, we hope to automate the selection of good parameter settings with respect to an individual class. We also intend to study further the interaction among factors and how it affects the selection of good values for the relevant parameters. Furthermore, preliminary research has suggested that credit to supporting correct decisions is often adequate, without any penalties at all. We plan to investigate this further, as well as the influence of credit for opposing incorrect decisions and penalties for failure to support correct ones. Meanwhile, we believe that RSWL represents a promising new approach to learning to manage a large body of heuristics.

Appendix

The concerns underlying ACE's heuristic tier-3 Advisors, drawn from the CSP literature are listed below. Each concern is computed dynamically unless otherwise indicated. One vertex is the neighbor of another if there is an edge between them. A *nearly neighbor* is the neighbor of a neighbor in the constraint graph. The *degree* of an edge is the sum of the degrees of the variables incident on it. The *edge degree* of a variable is the sum of edge degrees of the edges on which it is incident.

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

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

References

Bayardo, R. J. J. and R. Schrag (1996). Using CSP Look-Back Techniques to Solve Exceptionally Hard SAT Instances. *Principles and Practice of Constraint Programming CP-1996*, pp. 46-60.

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

Epstein, S. L. and E. Freuder (2001). Collaborative Learning for Constraint Solving. *Principles and Practice of Constraint Programming -- CP2001* (T. Walsh, Ed.), pp. 46 - 60, Springer 2001, Paphos, Cyprus.

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

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

Gent, I. P., P. Prosser and T. Walsh (1999). The Constrainedness of Search. *AAAI/IAAI* 1: 246-252.

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

Herbster, M. and M. Warmuth (1998). Tracking the Best Expert. *Machine Learning* 32: 151 - 178.

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

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

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

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

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

Russell, S. and P. Norvig (2003). *Artificial Intelligence A Modern Approach*, Prentice Hall, Upper Saddle River, NJ.

Sabin, D. and E. C. Freuder (1994). Contradicting Conventional Wisdom in Constraint Satisfaction. *Eleventh European Conference on Artificial Intelligence*, pp. 125-129, John Wiley & Sons, Amsterdam.

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