International Journal on Artificial Intelligence Tools
Vol. XX, No. X (2007) 1–17

World Scientific
www.worldscientific.com

RANDOM SUBSETS SUPPORT LEARNING A MIXTURE OF HEURISTICS

SMILJANA PETROVIC

Department of Computer Science, The Graduate Center of The City University of New York,
New York, NY, USA,
spetrovic@gc.cuny.edu

SUSAN L. EPSTEIN

Department of Computer Science, Hunter College and The Graduate Center of The City University of New York,
New York, NY, USA,
susan.epstein@hunter.cuny.edu

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Problem solvers, both human and machine, 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 mixture that works well on a specific class of problems. The principal result reported here is that randomly chosen subsets of heuristics can improve the identification of an appropriate mixture of heuristics. A self-supervised learner uses this method here to learn to solve constraint satisfaction problems quickly and effectively.

Keywords: mixture of heuristics, learning for constraint satisfaction

1. Introduction

Even within a familiar environment, well-trusted heuristics can vary dramatically in their performance on different problem classes. Some traditionally good heuristics actually perform quite poorly compared to their *duals* (opposites) on particular kinds of problems. Because it is difficult to predict which heuristics will perform best, we argue here for the selection of a mixture of heuristics from among a large, contradictory set.

Nonetheless, learning with a large, inconsistent set of heuristics presents considerable challenges. A self-supervised learner that gleans its training instances from problem traces requires solved problems. If such a learner is forced to work within a resource limit, and with many search heuristics of uncertain quality, it is put at a considerable disadvantage. Rather than use all the heuristics at once, our method consults a new, randomly-selected subset of them for each learning problem. As a result, the solver learns effective combinations of heuristics for challenging problems within a reasonable resource limit. The thesis of this work is that randomly chosen subsets from a large pool

of general, potentially inappropriate heuristics, can support the creation of a small, weighted mixture of heuristics that effectively guides search. Our principle result is a demonstration of this idea in heuristic-guided global search to solve constraint satisfaction problems.

After a review of constraint satisfaction and related work, we describe how mixtures of preference heuristics are learned. Subsequent sections describe the benefits of learning from subsets of a large set of heuristics, describe our experiments, and discuss the results.

2. Constraint satisfaction problems

A *constraint satisfaction problem* (*CSP*) is a set of variables, their associated domains, and a set of constraints, expressed as relations over subsets of those variables. An *instantiation* is an assignment of values to some 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 an instantiation of all its variables that satisfies all the constraints. A *binary CSP* has constraints on at most two variables. Such a 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.

Many real-world problems, such as planning and scheduling, graph-coloring, frequency assignment problems, image processing, language analysis and natural language understanding are modeled and solved as CSPs.[1,2] A *class* here is a set of CSPs with the same characterization. For example, binary CSPs in model B are characterized by $<n, m, d, t>$, where n is the number of variables, m the maximum domain size, d the density (fraction of edges out of $n(n-1)/2$ possible edges) and t the *tightness* (fraction of possible value pairs that each constraint excludes).[3] A class can also mandate some non-random structure on its problems. For example, a *composed problem* consists of a subgraph called its *central component* loosely joined to one or more subgraphs called *satellites*.[1]

A *global* search algorithm extends a partial instantiation with an assignment to a variable that is consistent with all previously assigned values. If an inconsistency arises, another value from the domain of that variable is tried. If all values in the domain of the variable are inconsistent, the current instantiation cannot be extended to a solution. In that case, search returns to an earlier consistent instantiation. During global search, *variable-ordering heuristics* select the next variable to be assigned a value and *value-ordering heuristics* select the value to assign to that variable. Heuristics can be *static* (the order is prespecified before search begins) or *dynamic* (the order depends upon the current state during search). For example, a *minimal static degree* heuristic selects a variable with the fewest constraints on it in the original problem, while a *minimal dynamic degree* heuristic selects a variable that, in the current state, is constrained by the fewest uninstantiated variables.

3. Related work

The idea that a mixture of experts can outperform a single expert goes back at least to Marquis de Condorcet (1745-1794). 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 other reasons for preferring a mixture of experts.[4] 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.

In supervised, on-line learning, where the best expert is unknown, under the worst-case assumption, *mixture of experts* algorithms have been proved asymptotically close to the behavior of the best expert.[5] In an off-line setting, there are many theoretical and experimental confirmations of the superiority of mixtures of classifiers (particularly decision trees and neural networks).[6,7,8] Hansen and Salamon proved that ensemble of accurate and diverse classifiers is more accurate than any of its individual classifiers.[9] As a result, methods that create an ensemble of classifiers that disagree in their predictions are particularly successful. The most popular algorithm of this type, AdaBoost, seeks a classifier that is better on examples for which the current ensemble's performance is poor.[10,11] Recent applications of mixture of experts include handwriting recognition and protein structure prediction.[12,13]

Multi-TAC learns a mixture of CSP heuristics.[14] Its heuristics are used one at a time to solve problems. Then, the best heuristic is chosen, combined with each of the others in turn so that the second serves only as a tie-breaker (rather than as part of the mixture, as in the work reported here). The process creates several layers of tie-breakers, until no further improvement is obtained.

The program described here faces challenges that do not arise when combining classifiers. Without an instructor, ACE must solve a CSP problem itself, to obtain training instances from which it can learn. On hard problems, only a reasonable mixture of experts will be able to solve the first such problem. To the best of our knowledge, the method described here, where only a random subset of available experts make decisions and are judged accordingly, is novel.

Randomization has been used in other ways to support in CSP search. Randomly selected points diversify local search to try to escape local minima.[15] Global search restarts with a degree of randomness to compensate for heavy tails in the search cost distribution.[16] Randomness can also be introduced to break ties in variable and value selection, to decide whether to apply inference procedures after a value assignment, or to select a backtrack point.[17,18]

4. Solving with a mixture of heuristics

ACE (the Adaptive Constraint Engine) learns to solve a class of CSPs. ACE is based on FORR, an architecture for the development of expertise from multiple heuristics.[19] ACE customizes a weighted mixture of heuristics for each problem class.[20] Heuristics are implemented by procedures called *Advisors*.

The search algorithm (in Figure 1) alternately selects a variable and then selects a value for it from its domain. The size of the resultant search tree depends upon the order in which values and variables are selected. Each Advisor can *comment* upon any number of choices (variables or values), and each comment has a *strength* that indicates its degree of support for the choice. For example, if an Advisor is given choices {A, B, C} and comments {(10, A), (9, C)}, it prefers A to C and does not recommend B. The Advisors referenced in this paper are described in Appendix A.

Search (p, $\mathcal{A}_{var}$, $\mathcal{A}_{val}$)

Until problem *p* is solved or the allocated resources are exhausted

Select unvalued variable *v*

$$v = \arg\max_{c \in V} \sum_{A \in \mathcal{A}_{var}} w(A) \cdot s(A,c)$$

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

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

Revise domains of all unvalued variables **inference**

Unless domains of all unvalued variables are non-empty

return to a previous alternative value **retraction**

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

After a value assignment, some form of *inference* detects values that are incompatible with the current instantiation. Here we use the MAC-3 algorithm to maintain arc consistency during search.[21] MAC-3 temporarily removes currently unsupportable values to calculate *dynamic domains* that are consistent with the current partial instantiation. If every value in any variable's domain is inconsistent, the current partial instantiation cannot be extended to a solution, so some *retraction* method is applied. Here we use *chronological backtracking*: the subtree rooted at an inconsistent node is pruned, and the most recent value assignment(s) withdrawn.

ACE's 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. Tier-1 Advisors are consulted in a user-specified order. Tier-2 Advisors address subgoals, but they are outside the scope of this paper. The decision-making described here focuses on the

heuristic Advisors in tier 3. When a decision is relegated 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 over all the comments is executed.

Each tier-3 Advisor's heuristic view is based on a descriptive metric. For each metric, there is a dual pair of Advisors, one that favors smaller values for the metric and one that favors larger values. Typically, one Advisor from each pair is reported as a good heuristic in the CSP literature, but ACE implements both of them. Two *benchmark Advisors*, one for value selection and one for variable selection, generate random comments. They provide a lower bound on performance and are excluded from decision making.

5. Learning from search experience

ACE does a form of self-supervised reinforcement learning. The only information available to it 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. Moreover, without supervision, we must declare what constitutes correct decisions. Clearly, the value selection at the root of any digression is wrong. Although variable selections are always correct (because with correct value assignments, any variable ordering leads to a backtrack-free solution), a variable selection here is considered incorrect if the value assignment to that variable subsequently failed. Finally, 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.

Given a class of 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. ACE uses a weight-learning algorithm to update its *weight profile,* the set of weights for its tier-3 Advisors. As in Figure 2, the learner gleans training instances from its own (likely imperfect) successful searches, and uses them to refine its search algorithm before it continues to the next problem. *Positive training instances* are those made along an error-free path extracted from a solution trace. *Negative training instances* are value selections that lead to unsolvable subproblems (*digressions*), as well as variable selections whose subsequent value assignment fails. Decisions made within a digression are not considered.

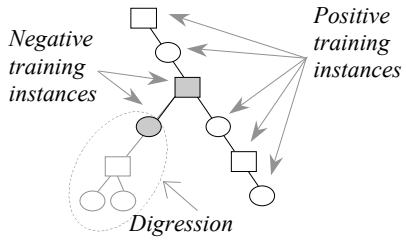

Fig. 2. The extraction of positive and negative training instances from the trace of a successful CSP search. Squares represent variable selections, and circles represent value selections.

The weight learning algorithm used here is a variation on *RSWL* (Relative Support Weight Learning), developed for ACE.[22] 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. RSWL calculates credits and penalties based on relative support. RSWL also estimates the *constrainedness* of the problem at the time of the training instance.[23] To reduce computational overhead here, we simply use the number of available choices. Our rationale is that the penalty for making an incorrect decision from among only a few choices should be larger than the penalty for selecting from among many choices.

6. The motivation for a large set of heuristics

The choice of appropriate heuristics from the many touted in the constraint literature is non-trivial. Even well-trusted individual heuristics vary dramatically in their performance on different classes. Consider, for example, the non-uniform performance in Table 1, as measured by average *steps* (variable selections or value selections). Traditional, off-the-shelf heuristics were used individually here on 50 problems from each of 3 challenging classes. *Max degree* does about half as much work as *Min domain* on the first two classes, but differs little on the third.

Table 1: Individual heuristic search performance on three classes of random problems.

Heuristic	***<30, 8, 0.26, 0.34>***	***<20, 30, 0.444, 0.5>***	***<50, 10, 0.38, 0.2>***
min domain	563.98	10,411.04	51,346.74
max degree	206.50	5,266.66	46,347.42
max forward degree	220.38	10,150.10	43,890.40
min domain/degree	233.64	4,194.02	35,174.52
max weighted degree	223.44	5,897.48	30,956.36
min dom/dynamic degree	210.82	3,941.54	30,791.36
min dom/weighted degree	204.58	4,089.84	30,024.66

Some traditionally good heuristics actually perform quite poorly compared to their *duals* (opposites) when the problems have non-random structure. Consider, for example, the performance of 3 pairs of duals on 50 composed problems in Table 2. Here, *Comp* problems have a central component in <22, 6, 0.6, 0.1>, one satellite in <8, 6, 0.72, 0.45>, and links between them with density 0.115 and tightness 0.05. The traditionally good heuristic *Max degree* fails to find any solution to 9 *Comp* problems within 100,000 steps, while its dual solves all the problems successfully. In several real-world problems a dual has also been shown superior to the traditional heuristic.[24,25,26] To achieve good performance, therefore, it is advisable to consider both the maximizing and minimizing versions of a heuristic's metric.

Table 2: Performance of traditionally good heuristics (in italics) and their duals on 50 composed problems.

Heuristic	***Percentage solved***	***Nodes***
max degree	*82%*	*19,901.76*
min degree	100%	64.60
max forward degree	*92%*	*10,590.64*
min forward degree	100%	64.50
min domain/degree	*86%*	*15,558.28*
max domain/degree	92%	10,922.82

A good combination of heuristics can outperform even the best individual heuristic, as Table 3 demonstrates. Indeed, a good pair of heuristics, one for variable selection and the other for value selection can perform significantly better than an individual one. The second line of Table 3 show that a good pair of heuristics, one for variable ordering and the other for value ordering, can perform significantly better than an individual heuristic. Nonetheless, the identification of such a pair is not trivial. For example, *max product domain value* better complements *min domain/dynamic degree* than it does *max weighted degree*. Finally, the last line of Table 3 demonstrates that combinations of more than two heuristics can further improve performance.

Table 3: The search performance of several mixtures of heuristics on three classes of challenging problems.

Mixture	***<30, 8, 0.26, 0.34>***	***<20, 30, 0.444, 0.5>***	***<50, 10, 0.38, 0.2>***
The best single heuristic from Table 1	204.58	3,941.54	30,024.66
min dom/dynamic degree + max product domain value	156.32	2,763.94	15,090.86
max weighted degree + max product domain value	179.20	3,891.56	22,273.80
Mixture found by ACE	141.38	2,501.74	12,401.26

7. Learning with one subset of Advisors at a time

As illustrated above, it is difficult to predict which heuristics will perform best on a set of problems. If one begins with a large initial list of heuristics, it probably contains many that perform poorly on a particular class of problems (*class-inappropriate heuristics*) and others that perform well (*class-appropriate heuristics*). On challenging problems, learning with all these heuristics presents two difficulties for the self-supervised learner. First, many class-inappropriate heuristics may combine to make bad choices, and thereby make it difficult to solve the problem within a reasonable step limit. Because only solved problems provide training instances for weight learning, no learning takes place until some problem is solved. Second, class-inappropriate heuristics occasionally acquire high weights when an initial problem is easy, and then control subsequent decisions, so that

either the problems go unsolved or the class-inappropriate heuristics receive additional rewards. ACE is able to *recover* (correct weights) gradually from such a situation, but recovery is faster under *full restart,* where ACE recognizes that its current learning attempt is not promising, abandons the responsible training problems, and restarts the entire learning process.[27] In this paper, ACE has recourse to full restart, but rarely resorts to it.

LearnFromRandomSubsets(C, $\mathcal{A}_{var}$, $\mathcal{A}_{val}$)

```
Initialize all weights to 0.05
Until termination of the learning phase
        Identify learning problem p in C
        Generate or accept x and y in [0,1]
        Randomly select subset S_var of x variable Advisors from A_var
        Randomly select subset S_val of y value Advisors from A_val
        Search (p, S_var , S_val)
        If p is solved
                for each training instance t from p
                for each Advisor A such that s(A, t) > 0
                        when t is a positive training instance
                                increase w(A)                   *reward*
                        when t is a negative training instance
                                decrease w(A)                   *penalize*
        else when full restart criteria are satisfied
        initialize all weights to 0.05
```

Fig. 3. Learning on a class *C* of problems with random subsets of variable Advisors from $\mathcal{A}_{var}$ and variable Advisors from $\mathcal{A}_{val}$. The *Search* algorithm is defined in Fig 1.

Learning with random subsets (in Figure 3) is a new approach that addresses both these issues in self-supervised learning. For each problem, ACE randomly selects a new subset of all the Advisors (here, a *random subset*), consults them, makes decisions based on their comments, and updates only their weights. Since global search is complete, under a large enough step limit, every problem used here will eventually be solved, regardless of the mixture and weight profile. There are, however, many factors that determine the number of steps required to find a solution. These include:

- The number and the degree of relevance of the participating heuristics.
- The problem difficultly, which may vary substantially within a given class for a given search algorithm.[28]
- The frequency with which the heuristics comment, their discriminative power, and their accuracy.[29] For instance, even appropriate heuristics must comment frequently if they are to direct decision making during search.

We address some of those factors further in Section 9. Meanwhile, to understand why learning with random subsets is effective, our premise 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 random subset, the problem is unlikely to be solved and no learning occurs. The repeated selection of a random subset for each new problem, however, will eventually produce some subset S with a majority of class-appropriate heuristics that solves its problem within a reasonable resource limit. As a result, all participating 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 thereby 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. In the rare case of repeated failure, RSWL resorts to full restart for recovery.

8. Experimental design

We tested learning with random subsets on three random CSP classes particularly difficult for their size (n and m). (Indeed, some of them appeared in the First International Constraint Solver Competition at CP-2005.) <50, 10, 0.38, 0.2> has many variables and relatively small domains. <20, 30, 0.444, 0.5> has fewer variables but larger domains. <30, 8, 0.31, 0.34> is hard for its size, but significantly easier than previous two. (Because their number of variables and domain sizes uniquely identifies classes used here, we refer to them henceforth simply by n-m.) We also tested on problems from *Comp*, defined in Section 6.

For ACE, a *learning phase* is a sequence of problems that it attempts to solve and from which it learns Advisors weights. A *testing phase* in ACE 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. During learning, failure on 8 of the last 10 tasks triggered a full restart of that learning phase. At most one full restart was permitted on each run. The counter for learning phase termination begins with the first solved problem, resets upon full restart, and terminates the phase when it reaches 30. Problems were not reused during learning, even with full restart. During the *testing phase,* ACE uses only those Advisors whose weight exceeds that of their respective benchmarks. For each problem class, every testing phase used the same 50 problems. (The same problems were also used in Tables 1 through 3.) The step limits on individual problems during learning were 100,000 for <50, 10, 0.38, 0.2>, 10,000 for <20, 30, 0.444, 0.5> and 1,000 for <30, 8, 0.31, 0.34> and *Comp*. The step limits during testing were 100,000 for <50, 10, 0.38, 0.2> and <20, 30, 0.444, 0.5>, 10.000 for <30, 8, 0.31, 0.34> and *Comp*. In all experiments, 3 tier-1 Advisors and some tier-3 Advisors are consulted. The initial set ("All" in the tables

below) includes 42 tier-3 Advisors, described in Appendix A: 28 for variable ordering and 14 for value ordering.

Three different ways to choose the heuristics applied to each problem were implemented:

- Use all the Advisors on every problem.
- Choose a fixed percentage f of the variable-ordering Advisors and f of the value-ordering Advisors, without replacement. We tested both $f = 0.3$ and $f = 0.7$.
- For each problem, select a random percentage r in [0.3, 0.7]. Choose r of the variable-ordering Advisors and r of the value-ordering Advisors, without replacement.

On harder problems, when all the Advisors were included on every learning problem, many consecutive problems went unsolved before one was solved and the first weights could be learned. We report here on the number of learning problems that went unsolved, and the number unsolved before any weights had been learned (*early failures*). We also report the average number of steps across all testing problems and the percentage of problems solved during testing (out of 50). Unless otherwise noted, results are averaged over 10 runs.

9. Results and discussion

Table 4 demonstrates the difficulties in learning on hard problems when an entire large body of heuristics is consulted. Here, learning is on exactly 30 problems. In the first run, the first 5 problems went unsolved, after which ACE learned weights and achieved good testing performance. The second run produced high weights for class inappropriate heuristics, triggered a full restart and then learned weights correctly, which again resulted in good testing performance. In the third run, however, the first 21 problems went unsolved. Early failures are not counted toward full restart — since the weights do not change, there is no need to reinitialize them. When the first solved problem in Run 3 produced inadequate weights, there were too few of the 30 problems left to trigger restart before the learning termination criterion was satisfied. As a result, testing performance was poor. We addressed this issue by revising the experimental design to learn on 30 problems from the first solved one. This improved performance, but was not enough to eliminate inadequate runs on harder classes.

Table 4: When all Advisors are consulted, learning on a fixed number of 20-30 problems may perform poorly.

	Learning			Testing	
Run	Problems	Unsolved	Early failures	Solved	Steps
Run 1	30	5	5	100%	2,842.94
Run 2	42	12	3	100%	2,808.96
Run 3	30	29	21	26%	86,043.40

Table 5 demonstrates the advantages of learning with random subsets. On 30-8 problems, early failures are rare, and with full restart ACE was able to learn appropriate weights even when all the Advisors were consulted. Performance on *Comp* was similar. This confirms that learning with random subsets does not harm performance on problem classes where it is unnecessary. On the more difficult 20-30 problems, and particularly on 50-10 problems, performance improved dramatically: there were no inadequate runs, and the percentage of unsolved problems and the percentage of early failures were reduced.

Table 5: Learning with different methods of selection for the participating Advisors subsets. (*) indicates that only 9 runs were completed.

		Learning			**Testing**	
Class	Advisors in learning	Problems	Unsolved	Early failures	Steps	Solved
30-8	All	37.90	6.60	1.80	185.75	100.00%
	Random 30%	31.50	2.40	1.50	186.47	100.00%
	Random 70%	33.40	2.90	0.50	185.20	100.00%
	Random 30%-70%	36.20	5.10	1.60	187.31	100.00%
Comp	All	30.00	2.40	0.00	773.48	94.00%
	Random 30%	30.00	1.40	0.00	670.39	94.40%
	Random 70%	30.10	1.80	1.00	602.62	95.60%
	Random 30%-70%	30.00	1.90	0.00	702.61	94.20%
20-30	All	42.80	14.80	11.00	6,195.68	98.60%
	Random 30%	32.40	8.80	2.40	2,798.22	100.00%
	Random 70%	32.30	4.00	2.30	2,769.37	100.00%
	Random 30%-70%	32.30	6.90	2.30	2,743.33	100.00%
50-10	All (*)	88.67	77.22	51.44	73,174.59	30.67%
	Random 30%	33.60	9.40	3.60	22,884.62	90.80%
	Random 70%	37.00	9.20	4.20	23,263.39	89.40%
	Random 30%-70%	33.70	6.10	3.70	23,855.27	87.60%

During learning, the time to select the next variable or a value for it is not necessarily directly proportional to the number of selected Advisors. This is primarily because a pair of Advisors that minimize and maximize the same metric 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. For example, the two Advisors whose metric is the ratio of dynamic domain size to weighted degree are relatively fast if those metrics were calculated earlier by other Advisors for their own use. Table 6 shows that

removing 70% of the Advisors saved only 40% of time per step. Removing 30% saved more than 20%, however, since more calculations could be reused. The reduction in total computation time is even more significant, because it incorporates the number of learning problems and steps per problem.

Table 6: Percentage of computation time during learning when random subsets are used, compared to computation time with all Advisors on the 20-30 class.

	Computation time per step	Learning time
All	100.00%	100.00%
Random 30%	60.29%	44.43%
Random 70%	78.75%	40.37%
Random 30%-70%	71.91%	45.73%

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

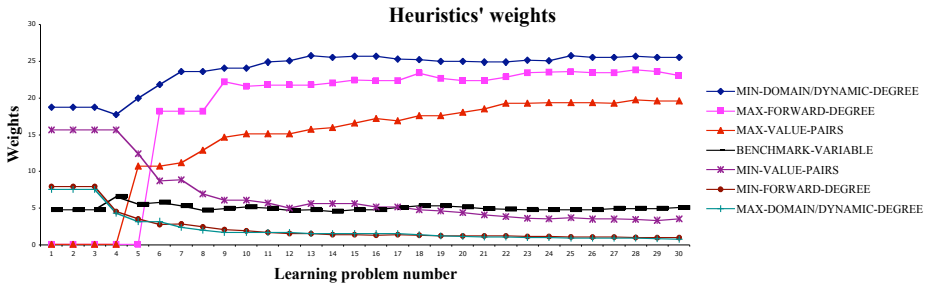

Fig. 4. Weights of selected Advisors during learning over 30 problems in a single run.

Table 7 demonstrates the robustness of learning with random subsets. In these experiments ACE was tested with fewer Advisors. Based on weights from successful runs in Table 4, the Advisors were first identified as class-appropriate or class-inappropriate for 20-30. ACE was then provided with three different sets of variable Advisors $\mathcal{A}_{var}$ in which class-inappropriate Advisors outnumbered class-appropriate ones. Both Set-1 and Set-2 were generated by the removal of 3 class-appropriate Advisors from 9 dual pairs. To create Set-1, one of the removed Advisors was the "neutral" *Max backward degree,* whose weight is typically low and not much higher than the weight of

its dual. To create Set-2, we removed two Advisors that typically earn high weights and lead decisions in a mixture (*Min Domain/Dynamic Degree* and *Max Weighted Degree*). Set-3 was biased even further toward class-inappropriate Advisors: it contained only 4 class-appropriate and 9 class-inappropriate Advisors. (The full list of Advisors in each set appears in Appendix B.)

Table 7: Learning when the pool of Advisors contains more class-inappropriate Advisors than class-appropriate.

			Learning			**Testing**	
	Advisors in learning	Runs completed	Problems	Unsolved	Early failures	Steps	Solved
Set 1	All	4	180.50	178.50	141.50	94,352.49	8.00%
	Random 30%	10	35.40	11.80	1.60	3,976.38	100.00%
	Random 70%	10	43.00	15.50	12.10	3,898.47	100.00%
	Random 30%-70%	10	35.70	10.20	4.80	3,874.91	100.00%
Set 2	All	3	225.00	223.00	186.00	94,361.65	8.00%
	Random 30%	10	44.30	23.20	5.80	3,847.50	100.00%
	Random 70%	10	58.00	33.50	22.10	5,084.70	99.60%
	Random 30%-70%	10	43.30	18.70	11.20	3,807.75	100.00%
Set 3	All	3	225.00	223.00	186.00	95,431.87	6.67%
	Random 30%	10	33.90	13.70	3.90	3,840.96	100.00%
	Random 70%	10	44.80	19.50	10.40	12,979.44	91.00%
	Random 30%-70%	10	34.90	12.40	3.60	12,835.01	91.00%

Clearly, when all provided Advisors were consulted, the predominance of class-inappropriate Advisors effectively prevented the solution of most problems. On the rare occasion that some (probably easy) problem was solved, the learned mixture was inappropriate, and failure persisted. With such extensive failures, even our library of 1000 problems was exhausted, so only 3 or 4 runs were completed under the "All" option. With random subsets, managing these sets of heuristics was possible: the class-appropriate heuristics were identified and appropriate weights were learned. Inadequate runs were rare, and only occurred when there were substantially fewer class-appropriate than class-inappropriate Advisors (Set 3).

Learning with random subsets was more successful (i.e., it produced fewer early and total failures) when Advisors were selected from Set 1 than from Set 2. This demonstrates that candidate heuristics relevant to the targeted class make learning easier. Nonetheless, even with a less powerful set of heuristics, learning with random subsets produced good mixtures.

One issue in learning with random subsets is how to determine a good subset size. Our experiments show that more early failures occur and more learning problems are required when random subsets are relatively large (70% vs. 30%). Intuitively, if there are few class-appropriate heuristics in $\mathcal{A}$, the probability that they are selected as a majority in a larger subset is small (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. As demonstrated in all three cases in Table 7, the number of early failures is significantly lower when the subset size is 30%, resulting in overall better performance.

When there are roughly as many class-appropriate as class-inappropriate Advisors (as in Table 5), the subset sizes are less important. In this case, the probability that any subset has more class-appropriate Advisors is near 0.5, and the expected number of trials is 2. In Table 5, ACE averaged fewer than 2 early failures on the 30-8 and *Comp* problems. Problems in 30-8 are easier and can be solved even without an overwhelming majority of class-appropriate Advisors. Due to the non-random structure of *Comp* problems, the difficultly of problems within a class vary enormously for most traditional individual CSP heuristics — some problems are extremely difficult while others are easy, so early failures are not an issue. More than two early failures on harder problems for a given step limit, however, suggest that a majority of class-appropriate Advisors is not enough; it require the selection of Advisors that are particularly good for the class.

Another observation from Tables 5 and 7 is that with larger subsets, there are few failures once the first problem is solved. When subsets are small, subsequent random subsets are less likely to overlap with those that proceeded them, and therefore less likely to include Advisors whose weights have been revised. As a result, failures occur even after some class-appropriate heuristics receive high weights. Thus, weights converge faster when subsets are larger. The data in Table 7 show that learning with random subsets of 70% requires not many more training problems after the first problem is solved.

Random selection of the subset size provides the benefits of both small and large random subsets. When the subset size varies, a smaller size makes it more likely that a subset with a more class-appropriate heuristics is selected early, while a larger size makes overlap more likely, and thereby speeds learning. All these experiments were successful

when the size of the random subset for each new learning problem was a random number within [0.3, 0.7].

10. Conclusion and future work

We have demonstrated that learning with random subsets performs significantly better than learning with a large set of Advisors. It manages a substantial set of heuristics, most of which may be class-inappropriate and contradictory. Better performance is indicated by fewer failures to solve a problem within the given step limit, faster decisions during learning, and more problems solved with fewer steps during testing. Learning with random subsets is also robust; it works well with different subset sizes, with different proportions of class-appropriate to class-inappropriate heuristics, and even with Advisors whose individual performance is relatively mediocre.

We are currently examining other parameters for learning with random subsets. These include the learning step limit, the termination criteria for learning, full restart parameters and the constrainedness of the problem class. We intend to use boosting to reuse problems and enforce learning from harder problems.[10] Particular attention will be paid to the initial set of Advisors and to synergies among them, given the recent results from factor analysis.[30] Another approach would be to reduce the degree of randomness as learning progresses.

Finally, learning with random subsets is demonstrated here on CSPs, but in principle it could be extended to other domains and to other kinds of experts participating in mixtures. Learning with random subsets relieves the user of the burden of providing domain specific knowledge for a solver. That is an important step toward automated problem solving.

Appendix A.

The following Advisors were used in these experiments.

Tier-1 Advisors:

- ***Victory.*** When only a single variable has no assigned value and has been selected, Victory comments in favor of any value in the dynamic domain of that variable.
- ***Degree Zero.*** When a variable is to be selected next, Degree Zero vetoes any variable whose dynamic degree is zero.
- ***Unique Value.*** When a variable is to be selected next, Unique Value forces the selection of any variable whose dynamic domain contains exactly one value.

Tier-3 Advisors:

The concerns underlying ACE's tier-3 Advisors are drawn from the CSP literature. (Citations are provided where possible.) Two vertices with an edge between them are *neighbors*. 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. Advisors are listed in dual pairs.

Variable selection Advisors:

- ***Min/Max Degree*** supports variables in increasing/decreasing order of their static (in the original constraint graph) degree.
- ***Min/Max Domain*** supports variables in increasing/decreasing order of their dynamic domain size after inference.
- ***Min/Max Domain/Degree*** supports variables in increasing/decreasing order of the ratio of their dynamic domain size to their static degree.[31]
- ***Min/Max Backward Degree*** supports variables in increasing/decreasing order of the number of their valued neighbors.
- ***Min/Max Forward Degree*** supports variables in increasing/decreasing order of their dynamic degree (number of unvalued neighbors).
- ***Min/Max Value Pairs*** supports variables in increasing/decreasing order of the number of pairs of values with their neighbors still supported by the current partial assignment.[32]
- ***Min/Max Static Connected Edges*** orders the edges in the original constraint graph descendingly, by the sum of the degrees of their vertices. They support variables in increasing/decreasing order of their incidence on these edges, with preference for the higher-degree vertex.
- ***Min/Max Static Less Connected Edges*** orders the edges in the original constraint graph ascendingly, by the sum of the degrees of their vertices. They support variables in increasing/decreasing order of their incidence on these edges, with preference for the higher-degree vertex.
- ***Min/Max Dynamic Connected Edges*** orders the edges in the dynamic constraint graph descendingly, by the sum of the degrees of their vertices. Supports variables in increasing/decreasing order of their incidence on these edges, with preference for the higher-degree vertex.
- ***Min/Max Dynamic Less Connected Edges*** orders the edges in the dynamic constraint graph ascendingly, by the sum of the degrees of their vertices. Supports variables in increasing/decreasing order of their incidence on these edges, with preference for the higher-degree vertex.
- ***Min/Max FF2*** supports variables in increasing/decreasing order of their likelihood of failure, computed by

$$\left(1 - \prod_{i \neq j} \left(1 - \text{tightness}(ij)^{|\text{domain}(j)|}\right)\right)^{|\text{domain}(i)|}$$

  They estimate *tightness(ij)* as the average tightness over the neighbors in the original constraint graph to the power of the original degree of the variable.[33]
- ***Min/Max Weighted Degree*** supports variables in increasing/decreasing order of the sum of the weights of the edges on which they are incident. Initially, edge weights are set to 1. Thereafter, each time inference from one endpoint of an edge wipes out the domain of the other endpoint, the weight of the edge is increased by 1.[34]
- ***Min/Max Domain/Weighted Degree*** supports variables in increasing/decreasing order of the ratio of their dynamic domain size to their weighted degree.[34]

Value selection Advisors:

- ***Min/Max Static Conflicts*** *Value* minimizes/maximizes based on the number of values that would be supported in the static domains of the unvalued neighbors of the variable.[35]
- ***Min/Max Small Domain Value*** minimizes/maximizes the domain size among the neighbors of the variable after this assignment.[35]
- ***Min/Max Product Domain Value*** minimizes/maximizes the product of the domain sizes of the neighbors of the variable after this assignment
- ***Min/Max Domain Score Value*** minimizes/maximizes the largest domain size of the neighbors of the variable after this assignment. It takes the number of unvalued variables with domains of that size as an exponent.
- ***Min/Max Weighted Domain Score Value*** minimizes/maximizes domain sizes of the neighbors of the variable after this assignment. It weights the number of future variables with domains of each size, a variant on an idea in.[35]
- ***Min/Max Secondary Pairs Value*** minimizes/maximizes the number of value pairs supported by this assignment from neighbors of the variable to nearly neighbors of the variable.
- ***Min/Max Secondary Value*** minimizes/maximizes the number of values supported by this assignment among nearly neighbors of the variable.

Appendix B.

Table 8: Advisors used for the experiments in Table 7.

	Class-appropriate Advisors	**Class-inappropriate Advisors**
Set-1	*Max-Degree* *Min-Domain* *Min-Domain/Dynamic-Degree* *Max-Weighted-Degree* *Min-Dynamic-Domain/Weighted-Degree* *Min-Static-Less-Connected-Edges*	*Min-Degree* *Max-Domain* *Max-Domain/Dynamic-Degree* *Min-Weighted-Degree* *Max-Dynamic-Domain/Weighted-Degree* *Min-Static-Connected-Edges* *Min-Backward-Degree* *Min-Forward-Degree* *Max-Domain/Degree*
Set-2	*Max-Degree* *Min-Domain* *Max-Backward-Degree* *Max-Forward-Degree* *Min-Dynamic-Domain/Weighted-Degree* *Min-Static-Less-Connected-Edges*	*Min-Degree* *Max-Domain* *Min-Backward-Degree* *Min-Forward-Degree* *Max-Dynamic-Domain/Weighted-Degree* *Min-Static-Connected-Edges* *Max-Domain/Degree* *Max-Domain/Dynamic-Degree* *Min-Weighted-Degree*
Set-3	*Max-Degree* *Min-Domain* *Min-Dynamic-Domain/Weighted-Degree* *Min-Static-Less-Connected-Edges*	*Min-Degree* *Max-Domain* *Max-Dynamic-Domain/Weighted-Degree* *Min-Static-Connected-Edges* *Min-Backward-Degree* *Min-Forward-Degree* *Max-Domain/Degree* *Max-Domain/Dynamic-Degree* *Min-Weighted-Degree*

References

1. Aardal, K. I., S. P. M. v. Hoesel, A. M. C. A. Koster, C. Mannino and A. Sassano (2003). Models and solution techniques for frequency assignment problems. *4OR: A Quarterly Journal of Operations Research* 1(4): 261-317.
2. Getoor, L., G. Ottosson, M. Fromherz and B. Carlson (1997). Effective redundant constraint for online scheduling. *Fourteenth National Conference on Artificial Intelligence*, pp. 302-307, Providence, Rhode Island.
3. Gomes, C., C. Fernandez, B. Selman and C. Bessière (2004). Statistical Regimes Across Constrainedness Regions. *10th Conf. on Principles and Practice of Constraint Programming (CP-04)* (M. Wallace, Ed.), pp. 32-46, Springer, Toronto, Canada.
4. Dietterich, T. G. (2000). Ensemble methods in machine learning. *First International Workshop on Multiple Classifier Systems*, pp. 1-15, Cagliari, Italy.
5. Kivinen, J. and M. K. Warmuth (1999). Averaging expert predictions. *Computational Learning Theory: 4th European Conference (EuroCOLT '99)*, pp. 153--167, Springer, Berlin.
6. 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.
7. Opitz, D. and J. Shavlik (1996). Generating accurate and diverse members of a neural-network ensemble. Advances in Neural Information Processing Systems 8: 535-541.
8. Ali, K. and M. Pazzani (1996). Error reduction through learning multiple descriptions. Machine Learning 24: 173-202.
9. Hansen, L. and P. Salamon (1990). Neural network ensembles. IEEE Transactions on Pattern Analysis and Machine Intelligence 12: 993-1001.
10. Schapire, R. E. (1990). The strength of weak learnability. Machine Learning 5(2): 197--227.
11. Freund, Y. and R. Schapire (1996). Experiments with a new boosting algorithm. *Thirteenth International Conference on Machine Learning*, pp. 148-156.
12. Alimoglu, F. and E. Alpaydin (1997). Combining multiple representations and classifiers for pen-based handwritten digit recognition. *Fourth International Conference on Document Analysis and Recognition*, pp. 637-640, Ulm, Germany.
13. Zhang X, Mesirov J.P. and W. D.L. (1992). Hybrid system for protein secondary structure prediction. Journal of Molecular Biology 225: 1049-1063.
14. Minton, S., J. A. Allen, S. Wolfe and A. Philpot (1995). An Overview of Learning in the Multi-TAC System. *First International Joint Workshop on Artificial Intelligence and Operations Research*, Timberline, Oregon, USA.
15. Selman, B., H. Kautz and B. Cohen (1996). Local search strategies for satisfiability testing. DIMACS Series in Discrete Mathematics and Theoretical Computer Science 26: 521--532.
16. Gomes, C. P., B. Selman, N. Crato and H. Kautz (2000). Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. Journal of Automated Reasoning: 67–100.
17. Gomes, C. (2003). Complete Randomized Backtrack Search (survey). *Constraint and Integer Programming: Toward a Unified Methodology*, pp. 233-283, Kluwer, Milano.
18. Lynce, I., L. Baptista and J. Marques-Silva (2001). Stochastic systematic search algorithms for satisfiability. *LICS Workshop on Theory and Applications of Satisfiability Testing (LICS-SAT)*.
19. Epstein, S. L. (1994). For the Right Reasons: The FORR Architecture for Learning in a Skill Domain. Cognitive Science 18: 479-511.
20. Epstein, S. L., E. C. Freuder and R. Wallace (2005). Learning to Support Constraint Programmers. Computational Intelligence 21(4): 337-371.
21. Sabin, D. and E. C. Freuder (1997). Understanding and Improving the MAC Algorithm. Principles and Practice of Constraint Programming: 167-181.
22. Petrovic, S. and S. L. Epstein (2006). Relative Support Weight Learning for Constraint Solving. *AAAI Workshop on Learning for Search*, pp. 115-122, Boston.

23. Gent, I. P., P. Prosser and T. Walsh (1999). The Constrainedness of Search. AAAI/IAAI 1: 246-252.
24. Petrie, K. E. and B. M. Smith (2003). Symmetry breaking in graceful graphs. *Principles and Practice of Constraint Programming CP-2005*, pp. 930--934, LNCS 2833.
25. Otten, L., M. Grönkvist and D. P. Dubhashi (2006). Randomization in Constraint Programming for Airline Planning. *Principles and Practice of Constraint Programming CP-2006*, pp. 406-420, Nantes, France.
26. Lecoutre, C., F. Boussemart and F. Hemery (2004). Backjump-based techniques versus conflict directed heuristics. *ICTAI*,: 549–557.
27. Petrovic, S. and S. L. Epstein (2006). Full Restart Speeds Learning. *Proceedings of the 19th International FLAIRS Conference (FLAIRS-06)*, Melbourne Beach, Florida.
28. 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.
29. Epstein, S. L. (2004). Metaknowledge for Autonomous Systems. *Proceedings of AAAI Spring Symposium on Knowledge Representation and Ontology for Autonomous Systems. AAAI*, pp. 61-68.
30. Wallace, R. J. (2006). Analysis of heuristic synergies. In Joint ERCIM/CologNet International Workshop on Constraint Solving and Constraint Logic Programming.
31. Bessière, C. and J.-C. Régin (1996). MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In "Principles and Practice of Constraint Programming - CP96, LNCS 1118" (E. C. Freuder, Ed.), pp. 61-75, Springer-Verlag.
32. Kiziltan, Z., P. Flener and B. Hnich (2001). Towards Inferring Labelling Heuristics for CSP Application Domains. *KI'01*, Springer-Verlag.
33. Smith, B. and S. Grant (1998). Trying Harder to Fail First. *European Conference on Artificial Intelligence*, pp. 249-253.
34. Boussemart, F., F. Hemery, C. Lecoutre and L. Sais (2004). Boosting systematic search by weighting constraints. *Sixteenth European Conference on Artificial Intelligence-ECAI'04*, pp. 146–150.
35. Frost, D. and R. Dechter (1995). Look-ahead Value Ordering for Constraint Satisfaction Problems. *IJCAI-95*, pp. 572-278.