Random Subsets Support Learning a Mixture of Heuristics

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

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

Abstract

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.

Introduction

A good combination of heuristics can outperform even the best individual heuristic. Even well-trusted individual heuristics vary dramatically on their performance on different classes. Some traditionally good heuristics actually perform quite poorly compared to their *duals* (opposites) when the problems have specific structure. It is difficult to predict which heuristics will perform best on a set of problems. Hence, we argue 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 under 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 potential benefits of learning from subsets of a large set of heuristics, describe our experiments, and discuss the results.

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. 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. Constraint satisfaction search heuristics either select the next variable to be assigned a value (*variable-ordering heuristics*) or the next value to assign to that variable (*value-ordering heuristics*).

A *class* is a set of CSPs with the same characterization. For example, binary CSPs in model B are characterized by $\langle n, m, d, t \rangle$, where n is the number of variables, m the maximum domain size, d the density (fraction of 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 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* (Aardal, Hoesel, et al., 2003).

Finding a CSP solution is an NP-complete problem; the worst-case cost is exponential in n for any known algorithm. Often, however, a CSP can be solved with a cost much smaller than the worst case. CSPs in the same class are ostensibly similar, but there is evidence that their difficultly may vary substantially for a given search algorithm (Hulubei and O'Sullivan, 2005).

Related work

There are several reasons why a combination of heuristics can offer improved performance, compared to a single expert (Valentini and Masulli, 2002). There may be no single heuristic that is best on all the problems. A combination of heuristics could thus enhance the accuracy and reliability of the overall system. On limited training data, different candidate heuristics may appear equally accurate. In this case, one could better approximate the unknown, correct heuristic by averaging or mixing candidates, rather than selecting one (Dietterich, 2000). The performance of algorithms that combine experts (equivalent in this context to

heuristics) has been theoretically analyzed for supervised learning compared to the best individual expert (Kivinen and Warmuth, 1999). Under the worst-case assumption, even when the best expert is unknown (in an online setting), mixture-of-experts algorithms have been proved asymptotically close to the behavior of the best expert (Kivinen and Warmuth, 1999).

Randomization is common in CSP search. Randomly selected points diversify local search to try to escape local minima (Selman, et al., 1996). Global search restarts with a degree of randomness to compensate for heavy tails in the search cost distribution (Gomes, Selman, et al., 2000). Randomness can 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 (Gomes, 2003; Lynce, Baptista, et al., 2001).

Solving with a mixture of heuristics

When *ACE* (the Adaptive Constraint Engine) learns to solve a class of CSPs, it customizes a weighted mixture of heuristics for the class (Epstein, Freuder, et al., 2005). ACE is based on FORR, an architecture for the development of expertise from multiple heuristics (Epstein, 1994). 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 {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 the appendix.

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 (Sabin and Freuder, 1994). MAC-3 temporarily removes currently unsupportable values to calculate *dynamic domains* that reflect the current instantiation. If every value in any variable's domain is inconsistent, the current partial instantiation cannot be extended to a solution, so some *retraction* method is applied. Here we use *chronological backtracking*: the subtree (*digression*) rooted at an inconsistent node is pruned, and the most recent value assignment(s) withdrawn.

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

Until problem *p* is solved or resources exhausted

Select unvalued variable *v*

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

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

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

Correct domains of all unvalued variables **inference**

Unless domains of all unvalued variables are non-empty

return to a previous alternative value **retraction**

Figure 1: Search with a weighted mixture of heuristics from $\mathcal{A}$. *w(A)* is the weight of Advisor *A; s(A, v)* is the support of Advisor *A* for a choice *v*.

ACE's Advisors are organized into three tiers. The decision-making described here focuses on the heuristic Advisors in tier 3. 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 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 as a lower bound for performance and are excluded from decision making.

Learning from search experience

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 at the root of a digression, as well as variable selections whose subsequent value assignment fails. Decisions made within a digression are not considered.

ACE does a form of self-supervised reinforcement learning. The only information available to it comes from

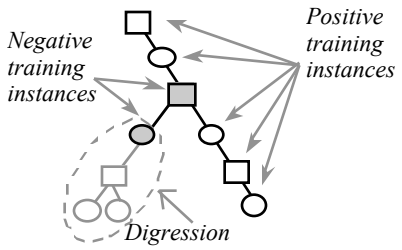

Figure 2: The extraction of positive and negative training instances from the trace of a successful CSP search. Squares are variable selections, circles are value selections.

its limited experience as it finds one solution to a problem. This approach is problematic for several reasons. Without supervision, we must somehow declare what constitutes correct decisions. Clearly, the value selection at the root of any digression is wrong. (Of course, that failure may be the result of some earlier decision.) Given correct value selections, however, any variable ordering can produce a backtrack-free solution. Thus the quality of a variable selection really reflects the subsequent search performance. Here, a variable selection is considered correct if no value assignment to that variable subsequently failed. Moreover, there is no guarantee that some other solution could not be found much faster, if even a single decision were different. 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.

The weight learning algorithm used here is a variation on *RSWL* (Relative Support Weight Learning), developed for ACE (Petrovic and Epstein, 2006b). 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 how constrained the problem was at the time of the training instance, in the spirit of (Gent, Prosser, et al., 1999). 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 from among many.

Motivation for using 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 on 20 problems from each of 2 challenging classes. (Definitions for all heuristics appear in the appendix.) Max degree does about half as much work as Min domain on the first class, but more work on the second.

Table 1: Individual heuristic search performance on two classes of challenging problems.

Individual heuristics	**<20, 30, .444, .5>** Steps	**<50, 10, .38, .8>** Steps
Min domain	14,404.00	52,387.30
Max degree	7,690.10	58,408.65
Max forward degree	16,119.85	48,364.15
Min dom/degree	5,337.60	41,033.70
Min dom/dynamic degree	4,989.15	36,508.05
Min dom/weighted degree	5,325.75	36,212.40

Table 2: Performance of traditionally good heuristics (in italics) and their duals on 100 composed class C problems.

Heuristic	**Unsolved problems**	**Steps**
Max degree	*19*	*2,117.03*
Min degree	0	64.72
Max forward-degree	*10*	*1,144.42*
Min forward-degree	0	64.49
Min domain/degree	*17*	*1,964.40*
Max domain/degree	9	1,000.88

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 100 class C problems in Table 2. Here, class C problems are composed, with 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 19 problems within 10,000 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 (Petrie and Smith, 2003; Otten, Grönkvist, et al., 2006; Lecoutre, Boussemart, et al., 2004). To achieve good performance, therefore, it is advisable to consider both the maximizing and minimizing versions of a heuristic's metric.

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. (Note too that some combinations of traditionally good heuristics are far better than others.) The last line of Table 3 demonstrates that combinations of more than two heuristics can further improve performance.

Updating one subset of Advisors at a time

As illustrated, 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

Table 3: Mixture of heuristics search performance on two classes of challenging problems

Mixtures	**<20, 30, .444, .5>** Steps	**<50, 10, .38, .8>** Steps
Min dom/weighted degree + Max Product Domain Value	3,447.05	14,282.70
Min dom/ dynamic degree + Max Small Domain Value	3,923.35	31,445.40
A 12-heuristic mixture found by ACE	3,127.30	8,554.80

LearnFromRandomSubsets

```
Until termination of the learning phase
   Identify learning problem p
   Generate or accept x and y in [0,1]
   Randomly select subset S_var of x variable Advisors from A
   Randomly select subset S_val of y value Advisors from A
   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
```

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

(*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 (Petrovic and Epstein, 2006a). In this paper, ACE has recourse to full restart, but rarely resorts to it.

Learning from random subsets is a new approach (in Figure 3) 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. Our premise is that eventually a random subset with a majority of class-appropriate heuristics will solve some problem, and that further learning from random subsets will produce an adequate weight profile for challenging problems. Initially, all Advisors' weights are set to 0.05.

Under a reasonable resource limit, when class-inappropriate heuristics predominate in a random subset, the problem is unlikely to be solved and no learning will occur. When class-appropriate heuristics predominate in a random subset *S* and they solve the problem, all participating Advisors will have their weights adjusted. On the next problem, the new random subset *S*′ likely contains some new, low-weight Advisors 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, and will therefore heavily influence decisions during search. If *S* succeeded because it contained more class-appropriate than class-inappropriate heuristics, *S* ′ *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*′ *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.

Any reduction in overall computation time here results mostly from the exploration of fewer partial instantiations. Individual decision time is not directly proportional to the number of selected Advisors. This is primarily because a pair of Advisors that minimize or maximize the same metric share the same major computational cost: calculating their common metric. For example, the bulk of the work for Min Product Domain Value lies in its one-step lookahead: calculating 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, dynamic domain/weighted-degree is based on weighted degree and dynamic domain, and is therefore relatively inexpensive if those metrics are in use by other Advisors.

Experimental design

We tested learning from random subsets on two CSP classes that are particularly difficult for their size (*n* and *m*). (Indeed, some of them appeared in the First International Constraint Solver Competition at CP-2005.) <50,8,0.38,0.8> has many variables and relatively small domains; <20,30,0.444,0.5> has fewer variables but larger domains. We also tested on problems from class C (defined earlier) and the somewhat easier <30,8,0.31,0.34>.

A *run* in ACE is a learning phase followed by a testing phase. During learning, failure on 7 of the last 13 tasks triggered a full restart. At most 2 full restarts were permitted; after 20 problems that did not trigger a restart, the learning phase terminated. Problems were not reused during learning, even with full restart. During ACE's *testing phase* it attempts to solve a sequence of fresh problems with learning turned off, using only those Advisors whose weight exceeds that of their respective benchmarks. For each problem class, every testing phase used the same 20 problems. (The same problems were also used in Tables 1 and 3.) In <50,8,0.38,0.8>, the step limits on individual problems were 100,000 during both the learning and testing phases. For the problems in <20,30,0.444,0.5>, which are somewhat easier, the step limit was 10,000 during

learning, and 100,000 during testing.

In these experiments, ACE began with 42 tier-3 Advisors, described in the appendix: 28 for variable ordering and 14 for value ordering. There were three types of experiments for each class; each with a different way to choose the heuristics applied to each problem:

- 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 [0.2,0.8]. Choose *r* of the variable-ordering Advisors and *r* of the value-ordering Advisors, without replacement.

All results reported here are averaged over 5 runs.

Results

ACE with RSWL could find an initial solution and then go on to learn successfully both in composed class C and in <30,3,0.31,0.34>. They served only to confirm that learning with random subsets does not harm performance on problem classes where it is unnecessary. (Details omitted.)

In <50,3,0.38,0.8> and <20,30,0.444,0.5>, however, when all the Advisors were included on every learning problem, there were often many consecutive unsolved problems before the first weights could be learned. We report here on the percentage of learning problems that went unsolved, and the percentage unsolved before any weights had been learned (*early failures*). We also report the percentage of problems that went unsolved during testing (out of 20), and the number of steps across all testing problems.

Table 4 demonstrates the difficulties in learning on hard problems with a large body of heuristics, when all the Advisors are consulted. In the first, fourth and fifth runs, after some failures, the learner was able to solve a problem, learn weights and achieve good testing performance. The second run produced high weights for class inappropriate heuristics, and did not recover, so testing performance was poor. In the third run, not a single problem was solved during learning. With the resultant unweighted combination of all 42 Advisors, 65% of the testing problems went unsolved. Experiments with more frequent full restarts produced better overall performance, but still did not eliminate inadequate runs. (Data omitted.)

Table 4: When all Advisors are referenced, learning on <20,30,0.444,0.5> problems can perform poorly.

	Learning		Testing	
Run	Unsolved	Early failures	Unsolved	Steps
Run 1	30.00%	25.00%	0%	4,065.10
Run 2	87.50%	50.00%	90%	95,334.15
Run 3	100.00%	100.00%	65%	75,675.50
Run 4	40.74%	37.04%	0%	4,791.40
Run 5	4.76%	0.00%	0%	4,312.35

Table 5 demonstrates the advantages of learning from random subsets. When Advisors were randomly selected for each task, performance improved: there were no inadequate runs, and the percentage of unsolved problem and the percentage of early failures was reduced. Table 5 also indicates how the size of the random subsets affects performance in both classes. More learning problems go unsolved with smaller (30% vs. 70%) random subsets, probably because the overlap among random subsets is smaller.

On the harder <50,3,0.38,0.8> class, with random subsets of 70%, the relatively high average decision steps on testing problems was due to a single run in which the variable Advisors received weights very similar to those in successful runs, but the value Advisors did not recover from incorrect initial weights. Because the variable-ordering weight profile was good, enough learning problems were solved to complete the phase. This was the only occasion on which such behavior developed.

One issue with the use of random subsets is how to determine a good subset size. When a random subset is too small, even if class-appropriate heuristics receive high weights, subsequent random subsets may not include any of them. On the other hand, if we assume that there are many more class-inappropriate than class-appropriate heuristics, a random subset that is too large may never have a majority of class-appropriate heuristics, and thus rarely solve problems. The experiments that permitted the size of the random subset to vary within [0.2, 0.8] during learning show no negative effect on either learning or testing performance. Moreover, on <50,3,0.38,0.8> they produced the best testing performance. Our intuition is that, with varying subset size, the smaller subsets ensure that one with a majority of class-appropriate heuristics is eventually selected, while the large subsets give more Advisors an opportunity to participate.

Another issue is how to synchronize the influences of variable and value Advisors. There remains the possibility that highly weighted, class-appropriate variable Advisors solve the problem, but class-inappropriate value heuristics

Table 5: Learning with different participating Advisors subsets. (*) indicates that only 2 runs were completed.

		Learning		Testing	
Class	Subset size	Unsolved	Early failures	Unsolved	Steps
20 30 .444 .5	100%	52.60%	42.41%	31%	36,835.70
	30%	32.20%	11.89%	0 %	3,608.27
	70%	14.56%	9.48%	0 %	3,962.62
	20%-80%	24.37%	7.72%	0 %	3,888.62
50 10 .38 .8	100% (*)	93.38%	58.10%	97.5%	97,972.85
	30%	31.64%	26.63%	1 %	15,599.71
	70%	26.45%	23.27%	10 %	23,499.99
	20%-80%	27.43%	20.29%	0%	13,206.32

are also reinforced.

We are currently examining other parameters for learning from random subsets. These include the learning step limit, the termination criteria for learning, full restart parameters and the constrainedness of the problem class. Particular attention will be paid to the size of the initial set of Advisors and synergies among them, given the "promise" and "fail-first" policies studied by (Wallace, 2006). Another approach would be to select Advisors probabilistically, based on their current weights. This would increase the likelihood that successful Advisors would be reselected. We intend to use boosting to ensure that learning is not limited to easy problems (Schapire, 1990).

We also plan to explore ways to eliminate underperforming Advisors during learning. Meanwhile, we have demonstrated that random subset learning 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. Learning with random subsets relieves the user of the burden of selecting search heuristics for a solver, an important step toward automated problem solving.

Appendix

The metrics underlying ACE's heuristic tier-3 Advisors were drawn from the CSP literature. Each metric produces a dual pair of Advisors. All are computed dynamically, except where noted. The *degree of an edge* is the sum of the degrees of the variables incident on it.

Variable selection metrics were static degree, dynamic domain size, FF2 (Smith and Grant, 1998), dynamic degree, number of valued neighbors, ratio of dynamic domain size to dynamic degree, ratio of dynamic domain size to degree, number of acceptable constraint pairs, *edge degree* (sum of degrees of the edges on which it is incident) with preference for the higher/lower degree endpoint, weighted degree, and ratio of dynamic domain size to weighted degree (Boussemart, Hemery, et al., 2004).

Value selection metrics were number of value pairs on the selected variable that include this value, and, for each potential value assignment: minimal resulting domain size among neighbors, number of value pairs from neighbors to their neighbors, number of values among neighbors of neighbors, neighbors' domain size, a weighted function of neighbors' domain size, and the product of the neighbors' domain sizes.

References

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.

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

Dietterich, T.G. (2000). Ensemble methods in machine learning. *CP-2000*, 1-15 Cagliari, Italy,

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

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

Gent, I. P., P. Prosser and T. Walsh (1996). The Constrainedness of Search. *AAAI/IAAI* 1: 246-252.

Gomes, C. (2003). Complete Randomized Backtrack Search. *In Constraint and Integer Programming: Toward a Unified Methodology*, pp. 233-283, Kluwer.

Gomes, C., C. Fernandez, B. Selman and C. Bessiere (2004). Statistical Regimes Across Constrainedness Regions. *CP-2004*, pp. 32-46, Springer, Toronto, Canada.

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.

Hulubei, T. and B. O'Sullivan (2005). Search Heuristics and Heavy-Tailed Behavior. *CP-2005*, pp. 328-342.

Kivinen, J. and M. K. Warmuth (1999). Averaging expert predictions. *EuroCOLT-99*, pp. 153-167.

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

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 2001)*.

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

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

Petrovic, S. and S. L. Epstein (2006a). Full Restart Speeds Learning. *FLAIRS-06*, Melbourne Beach, Florida.

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

Sabin, D. and E. C. Freuder (1994). Contradicting Conventional Wisdom in Constraint Satisfaction. *ECAI-1994*, pp. 125-129.

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

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.

Smith, B. and S. Grant (1998). Trying Harder to Fail First. *ECAI-1998*. pp. 249-253.

Valentini, G. and F. Masulli (2002). Ensembles of learning machines. *Neural Nets WIRN Vietri-02*

Wallace, R. J. (2006). Analysis of heuristic synergies. Recent Advances in Constraints. *Joint ERCIM/CologNet Workshop in Constraint Solving and Constraint Logic Programming*.