Learning to Solve Constraint Problems

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

[1]Department of Computer Science, The Graduate Center of 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

This paper explains why learning to solve constraint problems is so difficult, and describes a set of methods that has been effective on a broad variety of problem classes. The primary focus is on learning an effective search algorithm as a weighted subset of ordering heuristics. Experiments show the impact of several novel techniques on a variety of problems.

When a planning problem is cast as a constraint satisfaction problem (*CSP*), it can use the representational expressiveness and inference power inherent in constraint programming (Nareyek et al. 2005). If such an encoding lacks the necessary planning knowledge, a program might learn effective solution methods. Our thesis is that it is possible to learn to solve constraint satisfaction problems from experience. In this scenario, a program is given a set of CSPs and a set of search heuristics. It is then expected to learn an effective search algorithm, represented as a weighted combination of some subset of those heuristics.

From our perspective, the scenario's principal challenge is a plethora of purportedly "good" heuristics: heuristics to select variables or values, heuristics for inference, heuristics to determine when to restart. The focus here is on heuristics for traditional global search. After fundamental definitions and related work, this paper addresses the differences among search order heuristics, the power of mixtures of heuristics, and fundamental issues in learning to solve a class of CSPs. It then describes two algorithms for learning such mixtures, and additional learning methods that speed learning and often improve search performance.

Background and related work

A *CSP* is a set of variables, each with a domain of values, and a set of constraints, expressed as relations over subsets of those variables. CSP papers often present results on a *class* of CSPs, that is, a set of putatively similar problems. For example, a class of *model B* problems is 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) (Gomes et al. 2004). A problem 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 et al. 2003).

In a *binary* CSP, all constraints are on at most two variables. A binary CSP can be represented as a *constraint graph*, where vertices correspond to the variables (labeled by their domains), and each edge represents a constraint between its respective variables. Although the work reported here is on binary CSPs, in principle that is not a restriction.

A *solution* to a CSP is an instantiation of all its variables that satisfies all the constraints. Here, search for a solution iteratively selects a variable and assigns it a value from its domain, producing a *search node*. After each assignment, some form of *inference* detects values that are incompatible with the current instantiation. We use the MAC-3 inference algorithm to maintain arc consistency during search (Sabin et al. 1997). MAC-3 temporarily removes currently unsupportable values to calculate *dynamic domains* that reflect the current instantiation. If every value in any variable's domain is *inconsistent* (violates some constraint), then the current instantiation cannot be extended to a solution and some *retraction* method is applied. Retraction here is *chronological backtracking*: it prunes the subtree (*digression*) rooted at the inconsistent node and withdraws the most recent value assignment(s).

All data was generated with *ACE* (the Adaptive Constraint Engine). ACE learns a customized combination of pre-specified search heuristics for a class of CSPs (Epstein et al. 2005a). It attempts to solve some sequence of these problems within a specified resource limit (the *learning phase*). Then learning is turned off and the program attempts to solve a new sequence of problems drawn from the same set (the *testing phase*). A *run* is a learning phase followed by a testing phase. The resource limit is measured in *steps* (number of variable selections and value selections). The ability of a program to learn to solve CSPs is gauged here by the number of problems solved, the number of search steps, and the search tree size in nodes, averaged over a set of runs.

The premise of learning is that data can be calculated, stored, and applied to improve performance. Thus, it is reasonable to learn about how to solve a class of problems only if the effort expended, both to learn and to apply learned knowledge, can be justified by a frequent need to solve similar problems. For easy problems, learning is a waste of resources —the search algorithm should recognize and apply a simple, effective approach from its arsenal. On a class of more challenging problems, learning may be worthwhile if one expects to solve such problems often. Indeed, proponents of any new search algorithm inherently argue that most CSPs are enough like one another so that success on some set of classes, as described in their papers, bodes well for other classes yet untested. On a class of hard problems, learning is appealing because the given search algorithm is slower than one would wish, and thus the class is "hard" for the search algorithm at hand.

Historically most learning for constraint solving has been on an individual problem, rather than on an entire class. Such learning has primarily focused either on inconsistent partial instantiations that should be avoided or on constraints that provoke retraction (Dechter et al. 1987; Dechter 2003; Boussemart et al. 2004). Other work has learned weights for individual assignments (Refalo 2004), alternated among methods while solving an individual problem (Borrett et al. 1996), identified problematic edges with a preliminary local search (Ruml 2001; Eisenberg et al. 2003; Hoos et al. 2004), learned global constraints (Bessière et al. 2001; Bessière 2007), or addressed optimization problems and incomplete methods (Caseau et al. 1999; Caseau et al. 2004; Carchrae et al. 2005).

The argument for multiple heuristics

Despite enthusiasm for them in the CSP literature, *ordering heuristics* (those that select variables and values for them) display surprisingly uneven performance. Consider, for example, the performance of the variable selection heuristics in Table 1. (Definitions for all heuristics appear in the Appendix.) Even well-trusted individual heuristics such as these vary dramatically in their performance. For example, *max-weighted-degree* (Boussemart et al. 2004) is among the best individual heuristics when the number of variables is substantially larger than the maximum domain size (e.g., 50-10). It appears to be less effective, however, when there are more potential values than variables (e.g., 20-30).

Perhaps more surprising is that the opposite of a popular heuristic may be considerably more effective than the original. Let a *metric* be a function from a set of choices (variables or values) to the real numbers. A metric returns a *score* for each choice. An ordering heuristic is thus a preference for one extreme or the other of the scores

Table 1: Search tree size under individual heuristics on 50 problems from each of three randomly-generated Model B classes: <50, 10, 0.38, 0.2>, <20, 30, 0.444, 0.5>, and <30, 8, 0.26, 0.34> (referred to hereon as 50-10, 20-30, and 30-8, respectively).

Heuristic	***30-8***	***20-30***	***50-10***
min-domain	563	10,411	51,347
max-degree	206	5,267	46,347
max-forward-degree	220	10,150	43,890
min-domain/degree	234	4,194	35,175
max-weighted-degree	223	5,897	30,956
min-dom/dynamic-deg	211	3,942	30,791
min-dom/weighted-deg	205	4,090	30,025

Table 2: Performance of 3 popular heuristics (in italics) and their duals on 50 *Comp* problems (described in the text) under a 100,000-step limit. Observe how much better the duals perform on problems from this class.

Heuristic	Unsolved problems	Steps
Max degree	*9*	*19901.76*
Min degree	0	64.60
Max forward-degree	*4*	*10590.64*
Min forward-degree	0	64.50
Min domain/degree	*7*	*15558.28*
Max domain/degree	4	10922.82

returned by its metric. A *dual* for a heuristic reverses the import of its metric (e.g., *max-domain* is the dual of *min-domain*). Duals of popular heuristics may outperform them on real-world problems and on problems with non-random structure (Petrie et al. 2003; Lecoutre et al. 2004; Otten et al. 2006). For example, each composed problems in *Comp* has a Model B central component from <22, 6, 0.6, 0.1> linked to a single model B satellite from <8, 6, 0.72, 0.45> by edges with density 0.115 and tightness 0.05. The central component is substantially larger, with lower tightness and lower density than its satellite. These CSPs are particularly difficult for some traditional heuristics. For example, *max-degree* tends to select variables from the central component, while the decidedly untraditional *min-degree* tends to prefer variables from the satellite and thereby detects inconsistencies much earlier. Table 2 shows how three traditional heuristics and their duals fare on *Comp*. Surprisingly, the simplest duals do by far the best. This is of particular concern because the structural features of *Comp* often appear in real-world problems.

In practice, a good mixture of heuristics can outperform even the best individual one, as Table 3 demonstrates. The first line shows the best performance achieved by any traditional single heuristic from Table 1. The second line of Table 3 shows that a good pair of heuristics, one for variable ordering and the other for value ordering, can perform significantly better than an individual heuristic. Nonetheless, the identification of such a pair is not trivial. For example, *max-product-domain-value* better

Table 3: Search tree size under individual heuristics and under mixtures of heuristics on three classes of problems. ACE learns a different, high-performing mixture of more than two heuristics for each of these classes.

Mixture	30-8	20-30	50-10
The best heuristic from Table 1	205	3,942	30,025
Min dom/dynamic degree + Max Product Domain Value	156	2,764	15,091
Max-weighted-degree + Max Product Domain Value	179	3,892	22,273
Mixture found by ACE	141	2,502	12,120

complements *min-domain/dynamic-degree* than it does *max-weighted-degree*. The last line demonstrates that combinations of more than two heuristics can further improve performance.

Given these results, a program required to learn effective search without knowledge about problem structure should be provided with many popular heuristics, along with their duals. ACE's heuristics, each with its own metric, are gleaned from the CSP literature. To make a decision during search, ACE uses a weighted mixture of expressions of preference from a large number of such heuristics. This is a difficult task.

Why learning on a class of problems is hard

Without an instructor to provide examples of good and bad decisions, learning in our scenario is *self-supervised*, that is, the learner must assess both the quality of its own actions and the adequacy of its model of the environment. The <*search node, decision*> pairs from a solver's trace provide *self-generated* training instances. *Reinforcement learning* rewards or penalizes heuristics based on their ability to provide good search advice (Sutton & Barto, 1998), but in this context it faces a variety of difficulties.

A solution path may not provide good training instances. Since every variable must be assigned a value, any variable ordering must eventually lead to a solution if the problem is solvable. Nonetheless, some variable orders generate substantially fewer nodes, and may be more effective by several orders of magnitude; those are the ones we want our learner to produce. Self-generated training instances, however, may not necessarily represent good variable choices. Moreover, any variable ordering can lead to an error-free solution if each chosen value satisfies all constraints. As a result, the ease with which a solution is found is not a reliable criterion for evaluating the quality of the decisions that led to a solution.

The difficulty of a problem is hard to assess. Training instances must be drawn from the same population as testing instances, but a class of CSPs is only putatively similar. For a given search algorithm, in some circumstances the distribution of difficulty within a class is heavy tailed (Hulubei et al. 2005). Thus some problems will be extremely difficult, while others will be manageable, or even easy. When a learner confronts a CSP from a class, it is hard to predict how amenable the particular problem will be to the search algorithm. This issue arises whether or not the problems are "hard" in some fundamental sense. Variation in difficulty is not noise; it is inherent in the problems themselves and in their interaction with heuristics. In learning to solve CSPs, the skewed distribution within a problem class (as the result, perhaps, of an inappropriate heuristic) poses a particular challenge that is only exacerbated by more difficult classes.

The difficulty of a problem class is hard to assess. In Model B problems, for fixed values of n and m, there are value combinations for d and t that make the entire class of problems difficult in some fundamental sense (the *phase transition)* (Cheeseman et al. 1991). Even in a class at the phase transition (as are the classes in Table 3) there may be a wide range of difficulty, so that individual problems could give a misleading picture of the class as a whole. In theory one could assess the difficulty of a class using standard algorithms on a sample drawn from it, and thereby characterize the relative difficulty for problems with different parameter values. More generally, however, particularly in real-world contexts, this may not be possible to do beforehand. In such situations, the previously described difficulties of learning from learner-generated solution paths may be magnified.

The severity of an error is costly to assess. An *error* is a value assignment that is eventually retracted during search. Typically, even a handcrafted CSP solver arrives at a solution only after a lengthy series of errors. To penalize incorrect decisions appropriately, one should assess the severity of the error. Effectively, any incorrect decision creates an unsolvable problem. When a good solver errs, it will quickly discover its error. Gauging the effectiveness of error recovery, however, requires exploration of every possible ordering of value assignments in the digression, an unreasonable computational burden.

Errors may not be immediately apparent. An important issue in credit/blame assignment for reinforcement learning is that most retractions appear at some distance from the root of the search tree. In fact, even for hard but solvable problems, there are usually relatively few retractions at the top of the search tree, even with maintained arc consistency. Retractions often begin only after several decisions have been made. In such searches, the impact of bad decisions, especially variable selections, appears only after several more decisions have been made. As a result, it is difficult to assign blame to the true culprits.

Implications for learning. In summary, given a set of search traces, it is difficult to gauge how representative they are of effective search, difficult to identify sources of inefficiency from errors alone, and difficult to gauge how severe the errors are, how hard an individual problem is (despite its class designation), and even the degree to which a solution is based on good decisions. Moreover, since CSP solution is NP-complete, there can be no "gold standard" by which to judge the quality of a heuristic; the perfect search path must be assumed to be unobtainable on a regular basis. Clearly, for a program expected to learn an effective search algorithm based only on its own problem-solving experience, the interpretation of success and failure is not straightforward. Even the worst heuristic can solve some problems quickly. If such problems occur early in learning, then an ineffective heuristic will deceptively appear to be effective. If poor heuristics are reinforced early in learning, they will inevitably lead to poor performance on some subsequent problems.

Learning a mixture of heuristics

ACE, is based on FORR, an architecture for the development of expertise from multiple heuristics (Epstein

1994). ACE learns a customized weighted mixture of pre-specified heuristics for any given class. Guided by its ordering heuristics (here, *Advisors*) ACE solves problems in a given class and uses that experience to learn a *weight profile* (a set of weights for the Advisors). To select a variable or a value, ACE consults its Advisors. Each Advisor A_i expresses the *strength* s_{ij} of its preference for choice c_j. Based on the weight profile, the choice with the highest weighted sum of Advisors strengths to choices is selected:

$$\arg\max_j \sum_i w_i s_{ij} \qquad [1]$$

Initially, all weights are set to 0.05. During learning, ACE gleans training instances from its own (likely imperfect) successful searches. *Positive training instances* come from the error-free path to a solution. *Negative training instances* are incorrect value selections, as well as variable selections after which a value assignment fails. Decisions made within a digression are not considered. After each successful search, ACE extracts training instances from the trace, and updates the weight profile with a weight-learning algorithm before it goes on to the next problem.

Weights are based on the historical frequency with which an Advisor agreed with positive training instances and disagreed with negative ones, taken as a ratio because an Advisor may not always *comment* (express a preference) on a training instance. The learning algorithm presents each training instance, along with the possible actions available, to each Advisor. If an Advisor can discriminate among these actions by its comment strengths, its weight is adjusted: increased if it supports the correct decision, decreased otherwise. The actual weights are more than mere frequencies, however. A reward (the increment to the numerator) is not necessarily 1, and a penalty (the decrement to the numerator) can be substantial.

ACE has two effective algorithms that learn a weighted mixture of ordering heuristics for a class of CSPs. In *DWL* (Digression-based Weight Learning) an Advisor supports a decision if it gives that choice its highest rank. DWL rewards and penalizes search choices based on the number of nodes in the search tree, the size of its digressions, and performance on the preceding problems. *RSWL* (Relative Support Weight Learning) considers all heuristics' preferences when a decision is made. Weight reinforcements under RSWL depend upon the normalized difference between the strength the Advisor assigned to that decision and the average strength it assigned to all available choices (*relative support*). An Advisor supports a decision if its relative support is positive. Under RSWL, rewards and penalties are proportional to relative support.

Table 4: ACE's average steps to solution, with and without full restart. A lower step limit without full restart gives uneven performance.

Class	**<30, 8, 0.31, 0.34>**			**<30, 8, 0.18, 0.5>**		
Restart strategy	None	None	4 out of 7 failed	None	None	4 out of 7 failed
Learning step limit	20000	2000	500	10000	1000	500
Run 1	145.13	145.12	144.47	71.80	**3324.42**	73.00
Run 2	149.17	150.10	147.85	71.07	71.38	72.37
Run 3	163.28	**6541.17**	163.98	69.72	69.72	71.95
Run 4	146.85	151.63	152.73	70.85	70.43	73.23
Run 5	153.25	**6373.50**	156.27	71.53	71.92	71.97
Run 6	144.30	144.02	154.63	71.43	72.43	75.82
Run 7	154.90	157.73	158.10	72.37	71.42	71.50
Run 8	150.27	154.55	153.25	69.75	73.87	72.43
Run 9	135.93	157.68	162.58	71.25	**3370.53**	72.78
Run 10	150.77	154.25	158.00	69.90	71.62	73.20

Important learning mechanisms

The multitude of available CSP heuristics, their idiosyncratic applicability, and the issues described earlier drove the development of several important general learning mechanisms which we summarize here. Although these approaches are not limited to CSP, they are essential for good learning performance as envisioned here. All of the following experiments with ACE average results over 10 runs and use 42 Advisors, (described in the Appendix: 28 for variable ordering and 14 for value ordering).

Full restart. When class-inappropriate heuristics acquire high weights early in training, they often control the subsequent decisions and repeatedly fail to solve problems. *Full restart* recognizes that the current learning attempt is not promising, abandons the responsible training problems, and restarts the entire learning process with a freshly-initialized weight profile (Petrovic et al. 2006). Without full restart, reduced learning resources (the learning step limit) produce occasional unsatisfactory runs. With an appropriate full restart strategy, however, learning resources can be reduced by an order of magnitude without compromising performance. Full restart has proved most effective when it responds to the frequency of recent problem failure and when learning terminates after some number of consecutive solved problems. Table 4 demonstrates the power of restart. ACE monitored its own reliability during the learning phase: failure on 4 of the last 7 problems triggered a full restart. During the learning phase, ACE was required to try to solve 30 problems in its current full-restart attempt. Problems were never reused during learning, even under full restart.

Random subsets. Given an initial set of heuristics that is large and inconsistent, many class-inappropriate heuristics may combine to make bad choices, and thereby make it difficult to solve any problem within a given step limit. Because only solved problems provide training instances for weight learning, no learning can take place until some problem is solved. *Random subsets* have proved a successful approach to this issue: rather than consult all of its Advisors at once, ACE randomly selects a new subset of Advisors for each problem, consults them, makes decisions based on their comments, and updates only their weights (Petrovic et al. 2007b). During the experiments in Table 5, for each problem in the learning phase, *r* of the variable-

Table 5: Random subsets ($r < 100\%$) improve performance. (*) indicates that only 2 runs were completed.

		Learning		Testing	
Class	Subset size *r*	Unsolved	Early failures	Unsolved	Steps
20-30	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	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

ordering Advisors and *r* of the value-ordering Advisors were selected without replacement to make decisions during search on that problem. (For size 20%-80%, a random *r* in [.2,.8] was generated first.) Random subsets reduce *early failures* (problems unsolved before any weights were learned) during learning. They also reduce search tree size during testing, and increase the number of solved problems during both learning and testing, as shown in Table 5.

Fewer heuristics. A *benchmark* Advisor expresses random preferences over the same set of choices an Advisor faces. ACE has two such benchmarks, one for variable ordering and the other for value ordering. Benchmarks are excluded from decision-making, but weights are learned for them. To speed performance during testing, ACE uses only those Advisors whose learned weight exceeds that of their respective benchmarks. This typically eliminates about half the initial Advisors. We have experimented with further reductions in the number of Advisors during testing, as shown in Table 6. The more extensive reductions eventually increased search tree sizes for the 20-30 problems. For the 50-10 problems, however, the search tree size remained stable with a 31% speedup. We believe the explanation lies in the nature of the problems themselves. When there are many values compared to the number of variables, despite inference with MAC-3, domains remain large and many values still share the same scores (and

Table 6: Search tree size, number of solved problems, and time comparison with fewer Advisors during testing. Bold values are statistically significant performance reductions compared to the traditional benchmark approach (> bmk).

Var. Adv.	Val. Adv.	20-30			50-10		
		Steps	Solved	Time	Steps	Solved	Time
>bmk	>bmk	2,864	100%	100%	19,689	91%	100%
8	4	2,956	100%	71%	19,923	93%	84%
8	2	2,930	100%	55%	19,372	93%	70%
4	4	**3,198**	100%	87%	19,265	94%	77%
4	2	**3,176**	100%	67%	19,428	94%	69%

Table 7: Reduced search tree size with the full complement of learning methods described here, compared to the traditional single-heuristic approaches from Table 1. ACE values here are averaged over 10 runs; ACE Table 3 values are best individual runs.

Heuristic	*30-8*	*20-30*	*50-10*
min-domain	563	10,411	51,347
max-degree	206	5,267	46,347
max-forward-degree	220	10,150	43,890
min-domain/degree	234	4,194	35,175
max-weighted-degree	223	5,897	30,956
min-dom/dynamic-deg	211	3,942	30,791
min-dom/weighted-deg	205	4,090	30,025
ACE's learned mixture	**175**	**2,941**	**14,480**

strengths). With too few value-ordering Advisors, ties among value choices occur more often, so that random selection among tied values is more likely, making search decisions less prescient.

Borda-based voting. The metrics that underlie heuristics embody domain knowledge that reflects preferences among choices. Simple ranking ignores the degree of metric difference, linear interpolation attends to relative differences among scores, and exponential methods stress choices with higher scores while they reduce the influence of low-scoring choices dramatically. Two preference expression methods inspired by the Borda voting literature in political science (Brams et al. 2002) consider relative positions among scores and have proven particularly reliable (Petrovic et al. 2007a). When preference for fewer heuristics and full restart are combined with more sensitive expression of those Advisors' preferences, it is possible to significantly reduce both computation time and the size of the search tree on difficult problems.

Inference policy. Inference is intended to remove from consideration values that will not lead to a solution. An inference policy includes preprocessing, selection of an inference method, identification of relevant method parameters, and switching among methods. In pioneering work with ACE, we have shown the significant impact such a policy has on solution time, and that the choice of a good policy varies with both the problem class and the search order heuristics (Epstein et al. 2005b). We have also demonstrated how an inference policy can be learned automatically and can substantially improve performance. The most effective methods thus far are those that monitor and respond to domain changes after instantiations. They do less work than AC without reducing search performance.

Table 7 compares this combination with that of the single heuristics in Table 1. ACE's current development focuses on interleaving global with local search, and on a variety of structure-targeting representations that should continue to strengthen its ability to learn to search.

Appendix: Metrics for ACE's heuristics

Each metric produces two Advisors.

Metrics for variable selection were static degree, dynamic domain size, FF2, dynamic degree, number of valued neighbors, ratio of dynamic domain size to dynamic degree, ratio of dynamic domain size to degree, number of acceptable constraint pairs, static and dynamic edge degree with preference for the higher or lower degree endpoint, weighted degree, and ratio of dynamic domain size to weighted degree (Boussemart et al. 2004). Here, the *degree of an edge* is the sum of the degrees of its endpoints. The *edge degree* of a variable is the sum of edge degrees of the edges on which it is incident.

Metrics for value selection were number of value pairs for the selected variable that include this value, and, for each potential value assignment: minimum resulting domain size among neighbors, number of value pairs from neighbors to their neighbors, number of values among neighbors of neighbors, neighbors' domain size, a weighted function of neighbors' domain size, and the product of the neighbors' domain sizes. Two vertices with an edge between them are *neighbors*.

References

Aardal, K. I., S. P. M. van Hoesel, A. M. C. A. Koster, C. Mannino and A. Sassano (2003). "Models and solution techniques for frequency assignment problems." *4O* 1(4): 261-317.

Bessière, C. (2007). Learning Implied Global Constraints. In *Proceedings of IJCAI-2007*, Hyderabad, India.

Bessière, C. and J.-C. Régin (2001). Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI-2001*.

Borrett, J., E. Tsang and T. Walsh (1996). Adaptive constraint satisfaction. In *Proceedings of ECAI-96.*

Boussemart, F., F. Hemery, C. Lecoutre and L. Sais (2004). Boosting systematic search by weighting constraints. In *Proceedings of ECAI-2004*, IOS Press.

Brams, S. J. and P. C. Fishburn (2002). Voting procedures. Handbook of Social Choice and Welfare Volume 1**:** 173-236.

Carchrae, T. and J. C. Beck (2005). Cost-based Large Neighborhood Search In Proceedings of Workshop on the Combination of Metaheuristic and Local Search with Constraint Programming Techniques.

Caseau, Y., G. Silverstein and F. Laburthe (1999). A Meta-Heuristic Factory for Vehicle Routing Problems. In *Proceedings of CP-1999*, Springer Verlag.

Caseau, Y., G. Silverstein and F. Laburthe (2004). "Learning Hybrid Algorithms for Vehicle Routing Problems " *Theory and Practice of Logic Programming* 1(6): 779-806.

Cheeseman, P., B. Kanefsky and W. M. Taylor (1991). Where the REALLY Hard Problems Are. In *Proceedings of IJCAI-91*, Sidney, Australia.

Dechter, R. (2003). *Constraint Processing*. San Francisco, CA, Morgan Kaufmann.

Dechter, R. and J. Pearl (1987). "Network-based heuristics for constraint satisfaction problems." *Artificial Intelligence* 34: 1-38.

Eisenberg, C. and B. Faltings (2003). Using the Breakout Algorithm to Identify Hard and Unsolvable Subproblems. In *Proceedings of CP-2003*, Springer Verlag.

Epstein, S. L. (1994). "For the Right Reasons: The FORR Architecture for Learning in a Skill Domain." *Cognitive Science* 18(3): 479-511.

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

Epstein, S. L., E. C. Freuder, R. M. Wallace and X. Li (2005b). Learning Propagation Policies. In *Proceedings of Second International Workshop on Constraint Propagation and Implementation*, Sitges, Spain.

Gomes, C., C. Fernandez, B. Selman and C. Bessière (2004). Statistical Regimes Across Constrainedness Regions. In *Proceedings of CP- 2004*, Springer-Verlag.

Hoos, H. H. and T. Stützle (2004). *Stochastic Local Search: Foundations and Applications.* San Francisco, Morgan Kaufmann.

Hulubei, T. and B. O'Sullivan (2005). Search heuristics and heavy-tailed behavior. In *Proceedings of CP 2005*, Springer-Verlag.

Lecoutre, C., F. Boussemart and F. Hemery (2004). Backjump-based techniques versus conflict-directed heuristics. In *Proceedings of ICTAI-2004.*

Nareyek, A., E. C. Freuder, R. Fourer, E. Giunchiglia, R. P. Goldman, H. Kautz, et al. (2005). "Constraints and AI planning." *IEEE Intelligent Systems* 20(2): 62 - 72.

Otten, L., M. Grønkvist and D. P. Dubashi (2006). Randomization in Constraint Programming for Airline Planning. In *Proceedings of CP-2006*, Springer Verlag.

Petrie, K. E. and B. M. Smith (2003). Symmetry breaking in graceful graphs. In *Proceedings of CP-2003*, Kinsale, Ireland, Springer Verlag.

Petrovic, S. and S. L. Epstein (2006). Full Restart Speeds Learning. In *Proceedings of FLAIRS-2006.*

Petrovic, S. and S. L. Epstein (2007a). Preferences Improve Learning to Solve Constraint Problems. In *Proceedings of AAAI07 Workshop on Preference for Artificial Intelligence.*

Petrovic, S. and S. L. Epstein (2007b). Random Subsets Support Learning a Mixture of Heuristics. In *Proceedings of FLAIRS 2007*, Key West, AAAI.

Refalo, P. (2004). Impact-based search strategies for constraint programming. In *Proceedings of CP-2004.*

Ruml, W. (2001). Incomplete Tree Search using Adaptive Probing. In *Proceedings of IJCAI-2001.*

Sabin, D. and E. C. Freuder (1997). Understanding and Improving the MAC Algorithm. In *Proceedings of CP-97.*, Springer Verlag**:** 167-181.