Chapter 5
Learning a Mixture of Search Heuristics

Susan L. Epstein and Smiljana Petrovic

5.1 Introduction

An important goal of artificial intelligence research is to construct robust autonomous artifacts whose behavior becomes increasingly *expert*, that is, they perform a particular task faster and better than the rest of us [10]. This chapter explores the idea that, if one expert is a good decision maker, the combined recommendations of multiple experts will serve as an even better decision maker. The conjecture that a combination of decision makers will outperform an individual one dates at least from the Marquis de Condorcet (1745-1794) [50]. His Jury Theorem asserted 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.

It is difficult, or even impossible, to specify all the domain knowledge a program requires in a challenging domain. Thus an expert program must *learn*, that is, improve its behavior based on its own problem-solving experience. Machine learning algorithms extract their knowledge from *training examples,* models of desired behavior drawn from experience. Ideally, an oracle labels each training example as correct or incorrect, and the algorithm seeks to learn how to label not only those examples correctly, but also new *testing examples* drawn from the same population. Thus it is important that the learner not *overfit*, that is, not shape decisions so closely to the training examples that it performs less well on the remainder of the population from which they were drawn.

An *autonomous* learner has no oracle or teacher. It must monitor its own performance to direct its own learning, that is, it must create its own training examples

Susan L. Epstein
Department of Computer Science, Hunter College and The Graduate Center of The City University of New York, New York, New York, e-mail: susan.epstein@hunter.cuny.edu

Smiljana Petrovic
Department of Computer Science, Iona College, New Rochelle, New York, e-mail: spetrovic@iona.edu

and gauge its own performance. In addition, the learner must somehow infer the correct/incorrect labels for those examples from its own performance on the particular problem from the examples arose. Furthermore, the autonomous learner must evaluate its performance more generally: Is the program doing well? Has it learned enough? Should it start over?

Typically, one creates an autonomous learner because there is no oracle at hand. Such is the case in the search for solutions to constraint satisfaction problems (*CSPs*). The constraint literature is rich in heuristics to solve these problems, but the efficacy of an individual heuristic may vary dramatically with the kind of CSP it confronts, and even with the individual problem. A combination of heuristics seems a reasonable approach, but that combination must somehow be learned. This chapter begins, then, with general background on combinations of decision makers and machine learning, followed by specific work on combinations to guide CSP search. Subsequent sections describe ACE, an ambitious project that learns a mixture of heuristics to solve CSPs.

5.2 Machine Learning and Mixtures of Experts

In the face of uncertainty, a *prediction algorithm* draws upon theory and knowledge to forecast a correct decision. Often however, there may be multiple reasonable predictors, and it may be difficult for the system builder to select from among them. Dietterich gives several reasons not to make such a selection, that is, to have a machine learning algorithm employ a mixture of hypotheses [11]. On limited data, there may be different hypotheses that appear equally accurate. In that case, although one could approximate the unknown true hypothesis by the simplest one, averaging or mixing all of them together could produce a better approximation. Moreover, even if the target function cannot be represented by any of the individual hypotheses, their combination could produce an acceptable representation.

An *ensemble method* combines a set of individual hypotheses. There is substantial theoretical and empirical confirmation that the average case performance of an ensemble of hypotheses outperforms the best individual hypothesis, particularly if they are represented as decision trees or neural networks [2, 31, 46]. Indeed, for sufficiently accurate and diverse classifiers, the accuracy of an ensemble of classifiers has been shown to increase with the number of hypotheses it combines [22]. One well-known ensemble method is AdaBoost, which ultimately combines a sequence of learned hypotheses, emphasizing examples misclassified by previously generated hypotheses [14, 42].

More generally, a *mixture of experts algorithm* learns from a sequence of trials how to combine its experts' predictions [26]. In a supervised environment, a trial has three steps: the mixture algorithm receives predictions from each of e experts, makes its own prediction y based on them, and then receives the correct value y'. The objective is to create a mixture algorithm that minimizes the *loss function* (the distance between y and y'). The performance of such an algorithm is often mea-

sured by its *relative loss*: the additional loss to that incurred on the same example by the best individual expert. Under the worst-case assumption, mixture of experts algorithms have been proved asymptotically close to the behavior of the best expert [26].

5.3 Constraint Satisfaction and Heuristic Search

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. In a *binary* CSP, each constraint is on at most two variables. A *problem class* 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 constraints out of $n(n-1)/2$ possible constraints) and t the *tightness* (fraction of possible value pairs that each constraint excludes) [19]. 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.

A randomly generated problem class may also mandate a specific structure for its CSPs. For example, each of the *composed problems* used here consists of a subgraph (its *central component*) loosely joined to one or more additional subgraphs (its *satellites*) [1]. Figure 5.1 illustrates a composed problem with two satellites. Geometric CSPs also have non-random structure. A random geometric graph $<n, D>$ has n vertices, each represented by a random point in the unit square [25]. There is an edge between two vertices if and only if their (Euclidean) distance is no larger than D. A class of random geometric CSPs $<n, D, d, t>$ is based on a set of random geometric graphs $<n, D>$. In $<n, D, d, t>$, the variables represent random points, and constraints are on variables corresponding to points close to each other. Additional edges ensure that the graph is connected. Density and tightness are given by the parameters d and t, respectively. Figure 5.2 illustrates a geometric graph with 500 variables. Real-world CSPs typically display some non-random structure in their constraint graphs.

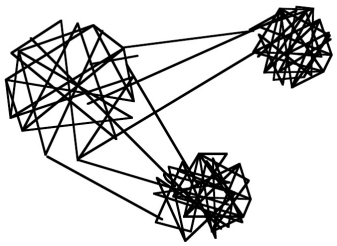

Fig. 5.1 Composed problem with two satellites

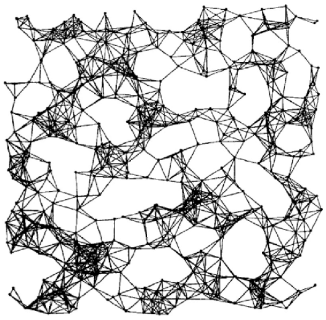

Fig. 5.2 Geometric graph from [25]

This chapter presents experiments on six CSP classes. *Geo* (the geometric problems <50, 10, 0.4, 0.82>) and *Comp* (the composed problems with central component in model B with <22, 6, 0.6, 0.1>, linked to a single model B satellite with <8, 6, 0.72, 0.45> by edges with density 0.115 and tightness 0.05) are classes of structured CSPs. The others are model B <50, 10, 0.38, 0.2>, which is exceptionally hard for its *size* (*n* and *m*); <50, 10, 0.18, 0.37>, which is the same size but somewhat easier; <20, 30, 0.444, 0.5>, whose problems have large domains; and <30, 8, 0.26, 0.34> which are easily compared to the other classes, but difficult for their size. Some of these problems appeared in the First International Constraint Solver Competition at CP-2005.

An instantiation assigns a value to all (*full* instantiation) or some (*partial* instantiation) of the variables. A *solution* to a CSP is a full instantiation that satisfies all the constraints. A *solvable* CSP has at least one solution. A solver *solves* a CSP if it either finds a solution or proves that it is *unsolvable,* that is, that it has no solution. All problems used in the experiments reported here are randomly generated, solvable binary CSPs with at least one solution.

Traditional (*global*) CSP search makes a sequence of decisions that instantiates the variables in a problem one at a time with values from their respective domains. After each value assignment, some form of *inference* detects values in the domains of *future variables* (those not yet instantiated) that are incompatible with the current instantiation. The work reported here uses the MAC-3 inference algorithm to maintain arc consistency during search [40]. MAC-3 temporarily removes currently unsupportable values to calculate *dynamic domains* that reflect the current instantiation. If, after inference, every value in some future variable's domain is *inconsistent* (violates some constraint), a *wipeout* has occurred and the current partial instantiation cannot be extended to a solution. At that point, some *retraction* method is applied. Here we use *chronological backtracking*, which prunes the subtree (*digression*) rooted at an inconsistent *node* (assignment of values to some subset of the variables) and withdraws the most recent value assignment(s).

The efficacy of a constraint solver is gauged by its ability to solve a problem, along with the computational resources (CPU time and search tree size in nodes) required to do so. Search for a CSP solution is NP-complete; the worst-case cost is exponential in the number of variables *n* for any known algorithm. Often, however,

a CSP can be solved with a cost much smaller than that of the worst case. A CSP *search algorithm* specifies heuristics for variable (and possibly value) selection, an inference method, and a backtracking method. It is also possible to *restart* search on a problem, beginning over with no assignments and choosing a new first variable-value pair for assignment.

In global search, there are two kinds of search decisions: select a variable or select a value for a variable. Constraint researchers have devised a broad range of variable-ordering and value-ordering heuristics to speed search. Each heuristic relies on its own *metric*, a measure that the heuristic either maximizes or minimizes when it makes a decision. *Min domain* and *max degree* are classic examples of these heuristics. (A full list of the metrics for the heuristics used in these experiments appears in the Appendix.) A metric may rely upon dynamic and/or learned knowledge. Each such heuristic may be seen as expressing a preference for choices based on the scores returned by its metric. As demonstrated in Section 5.5.1, however, no single heuristic is "best" on all CSP classes. Our research therefore seeks a combination of heuristics.

In the experiments reported here, resources were controlled with a *node limit* that imposed an upper bound on the number of assignments of a value to a variable during search on a given problem. Unless otherwise noted, the node limit per problem was 50,000 for <50, 10, 0.38, 0.2>; 20,000 for <20, 30, 0.444, 0.5>; 10,000 for <50, 10, 0.18, 0.37>; 500 for <30, 8, 0.26, 0.34>; and 5,000 for *Comp* and *Geo* problems. Performance was declared *inadequate* if at least ten out of 50 problems went unsolved under a specified resource limit.

A good mixture of heuristics can outperform even the best individual heuristic, as Table 5.1 demonstrates. The first line shows the best performance achieved by any traditional single heuristic we tested. The second line illustrates the performance of a random selection of heuristics, without any learning. (These experiments are non-deterministic and therefore averaged over a set of ten runs.) On one class, the sets of heuristics proved inadequate on every run, and on the other class, five runs were inadequate and the other five dramatically underperformed every other approach. The third line shows that a good pair of heuristics, one for variable ordering and the other for value ordering, can perform significantly better than an individual heuristic. The last line of Table 5.1 demonstrates that a customized combination of more than two heuristics, discovered with the methods described here, can further improve performance. Of course, the use of more than one heuristic may increase solution time, particularly on easier problems where a single heuristic may suffice. On harder problems, however, increased decision time is justified by the ability to solve more problems. This chapter addresses work on the automatic identification of such particularly effective mixtures.

Table 5.1: Search tree size under individual heuristics and under mixtures of heuristics on two classes of problems. Each class has its own particular combination of more than two heuristics that performs better

	<20, 30, 0.444, 0.5>			*<50, 10, 0.38, 0.2>*		
Guidance	***Nodes***	***Solved***	***Time***	***Nodes***	***Solved***	***Time***
Best individual heuristic tested	3,403.42	100%	10.70	17,399.06	84%	79.02
Randomly selected combination of more than two heuristics	five inadequate runs			ten inadequate runs		
Best pair of variable-ordering and value-ordering heuristic identified	1,988.10	100%	17.73	10,889.00	96%	76.16
Best learned weighted combination of more than 2 heuristics found by ACE	1,956.62	100%	29.22	8,559.66	98%	111.20

5.4 Search with More than One Heuristic

Many well-respected machine-learning methods have been applied to combine algorithms to solve constraint satisfaction problems. This work can be characterized along several dimensions: whether more than one algorithm is used on a single problem, whether the algorithms and heuristics are known or discovered, and whether they address a single problem or a class of problems.

5.4.1 Approaches that Begin with Known Algorithms and Heuristics

Given a set of available algorithms, one approach is to choose a single algorithm to solve an entire problem based upon experience with other problems (not necessarily in the same class). Case-based reasoning has been used this way for CSP search. A feature vector characterizes a solved CSP and points to a set of strategies (a model, a search algorithm, a variable-ordering heuristic, and a value-ordering heuristic) appropriate for that instance [17]. Given a new CSP, majority voting by strategies associated with similar CSPs chooses a strategy for the current one. In a more elaborate single-selection method, a Support Vector Machine (*SVM*) with a Gaussian kernel learns to select the best heuristic during search at checkpoints parameterized by the user [3]. Training instances are described by static features of a CSP, dynamic features of the current partial instantiation, and labels on each checkpoint that indicate whether each heuristic has a better runtime than the default heuristic there.

The solver applies the default heuristic initially but, after restart, replaces the default with a randomly selected heuristic chosen from those that the SVM preferred.

A slightly more complex approach permits a different individual heuristic to make the decisions at each step, again based on experience solving other problems not necessarily in the same class. A *hyperheuristic* decides which heuristic to apply at each decision step during search [7]. A hyperheuristic is a set of problem states, each labeled by a condition-action rule of the form "if the state has these properties, then apply this heuristic." A genetic algorithm evolves a hyperheuristic for a given set of states [45]. To solve a CSP, the best among the evolved hyperheuristics is chosen, and then the heuristic associated with the problem state most similar to the current partial instantiation is applied.

Alternatively, a solver can be built for a single CSP from a set of known search algorithms that take turns searching. For example, *REBA* (Reduced Exceptional Behavior Algorithm) applies more complex algorithms only to harder problems [5]. It begins search with a simple algorithm, and when there is no indication of progress, switches to a more complex algorithm. If necessary, this process can continue through a prespecified sequence of complex algorithms. The complexity ranking can be tailored to a given class of CSPs, and is usually based on the median cost of solution and an algorithm's sensitivity to exceptionally hard problems from the class. In general the most complex of these algorithms have better worst-case performance but a higher average cost when applied to classes with many easy problems that could be quickly solved by simpler algorithms. Another approach that alternates among solvers is CPHydra. It maintains a database of cases on not necessarily similar CSPs, indexed both by static problem features and by modeling selections [30]. Each case includes the time each solver available to CPHydra took to solve the problem. CPHydra retrieves the most similar cases and uses them to generate a schedule that interleaves the fastest solvers on those cases.

It is also possible to race algorithms against one another to solve a single problem. An *algorithm portfolio* selects a subset from among its available algorithms according to some schedule. Each of these algorithms is run in parallel to the others (or interleaved on a single processor with the same priority), until the fastest one solves the problem [21]. With a *dynamic algorithm portfolio*, schedule selection changes the proportion of CPU time allocated to each heuristic during search. Dynamic algorithm portfolios favor more promising algorithms [16] or improve average-case running time relative to the fastest individual solver [44]. A particularly successful example is SATzilla-07. It builds a portfolio for each *SAT* (propositional satisfiability) problem instance *online* as it searches [49]. Based on features of the instance and each algorithm's past performance, SATzilla-07 uses linear regression to build a computationally inexpensive model of empirical hardness that predicts each algorithm's runtime on a given SAT problem.

5.4.2 Approaches that Discover Their Own Algorithms

Other autonomous learners seek to discover their own heuristics, given inference and backtracking mechanisms. One approach is to discover a combination of existing algorithms appropriate for a class of problems. For example, Multi-TAC learns an ordered list of variable-ordering heuristics for a given class of CSPs [28]. Beginning with an initially empty list of heuristics, each remaining heuristic is attached to the parent to create a different child. The utility of a child is the number of instances it solves within a given time limit, with total time as a tiebreaker. The child with the highest utility becomes the new parent. This process repeats recursively until it does not produce any child that improves upon its parent. The resulting list is consulted in order. Whenever one heuristic cannot discriminate among the variables, those it ranks at the top are forwarded to the next heuristic on the list. Another way to link a class of problems to a combination of algorithms is to construct a class description based on the algorithms' performance. For example, one effort constructs and stores portfolios for quantified Boolean formulae that are generalizations of SAT problems [41].

Local search is an alternative paradigm to global search. *Local search* only considers full instantiations, and moves to another full instantiation that changes the value of some metric. Heuristics specify the metric and how to select among instantiations that qualify. Multiple search heuristics have been integrated with value-biased stochastic sampling [9]. On a given problem instance, multiple restarts sample the performance of different base heuristics. Then the program applies extreme value theory to construct solution quality distributions for each heuristic, and uses this information to bias the selection of a heuristic on subsequent iterations. Extensive research has also been conducted on optimization problems to learn which low-level local search heuristics should be applied in a region of the solution space [8]. One approach combined local search heuristics for SAT problems [29]. Each step chose a constraint to be adjusted based on some measure of its inconsistency in the current instantiation. Then a heuristic was selected probabilistically, based on its expected *utility value* (ability to ameliorate the violation of that constraint). All utility values were initially equal, and then positively or negatively reinforced based on the difference between the current total cost and the total cost the last time that constraint was selected.

The building blocks for a new algorithm need not be standalone algorithms themselves. For example, CLASS discovers local search variable-ordering heuristics for a class of SAT problems with a genetic algorithm on prespecified heuristic primitives [15]. Its primitives describe the currently satisfied clauses and search experience. CLASS uses them to construct LISP-like s-expressions that represent versions of standard local search variable-selection heuristics. Its initial population is a set of randomly generated expressions. Each population is scored by its performance on a set of problems. Then, to construct the next generation, instead of using traditional crossover and mutation, CLASS creates ten new children from its primitives and makes the ten lowest-scoring expressions less likely to survive. The best heuristic found during the course of the search is returned.

5.5 A Plan for Autonomous Search

As constraint satisfaction is increasingly applied to real-world problems, some work on mixtures of heuristics is more applicable than others. Local search, for example, will not halt on an unsolvable problem, and many real-world problems are unsolvable. Classes of problems recur, particularly in real-world environments (e.g., scheduling the same factory every week), and so it seems reasonable to learn to solve a class of CSPs rather than address each new problem in isolation. The success of a case-based method depends in large part on its index and ability to detect similarity, but real-world CSPs vary broadly, and that breadth challenges both the index and the similarity matching. Methods that learn heuristics are designed to replace the traditional heuristics, which perform far better on some problem classes than on others. Nonetheless, the research supporting these heuristics is extensive, and when they work, they work very well; discarding them seems premature.

The remainder of this chapter therefore uses global search on a class of similar CSPs with a large set of known heuristics. Although there are a great many well-tested heuristics, the successful methods that combine or interleave them have dealt with relatively few *candidates* (possible heuristics), despite the burgeoning literature. There are three challenges here: which candidates to consider for a mixture, how to extract training examples, and how autonomous search should gauge its own performance.

5.5.1 Candidate Heuristics

Learning is necessary for CSP solutions because even well-trusted individual heuristics vary dramatically in their performance on different classes. Consider, for example, the performance of five popular variable-ordering heuristics on the three classes in Table 5.2. *Min-domain/dynamic-degree* is the most successful on <20, 30, 0.444, 0.5> problems, but it is inadequate on *Comp* problems.

Table 5.2: Average number of nodes explored by traditional variable-ordering heuristics (with lexical value ordering) on 50 problems from each of three classes. The best (in bold) and the worst (in italics) performance by a single heuristic vary with the problem class. Problem classes were defined in Section 5.3

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

On real-world problems and on problems with non-random structure, the opposite of a traditional heuristic may provide better guidance during search [33, 27, 32]. The *dual* of a heuristic reverses the import of its metric (e.g., *max domain* is the dual of *min domain*). Table 5.3 demonstrates the superior performance of some duals on *Comp* problems. Recall that a *Comp* problem has a central component that is substantially larger, looser (has lower tightness), and sparser (has lower density) than its satellite. Once a solution to the subproblem defined by the satellite is found, it is relatively easy to extend that solution to the looser and sparser central component. In contrast, if one extends a partial solution for the subproblem defined by the central component to the satellite variables, inconsistencies eventually arise deep within the search tree. Despite the low density of the central component in such a problem, its variables' degrees are often larger than those in the significantly smaller satellite. The central component proves particularly attractive to two of the traditional heuristics in Table 5.2 , which then flounder there. We emphasize again that the characteristics of such composed problems are often found in real-world problems. Our approach, therefore, is to take as candidates many popular heuristics, along with their duals.

Table 5.3: Average number of nodes explored by three traditional heuristics (in italics) and their duals on *Comp* problems (described in Section 5.3). Note the better performance of two of the duals here

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

5.5.2 Training Examples

From a successful search, the solver extracts both positive and negative training examples. Each training example is the current instantiation and the decision made there. Here, too, difficulties arise. The search trace is not necessarily the best way to solve that CSP. There may have been a better (smaller search tree or less elapsed time) way to solve it. Thus the training examples selected may not be those an oracle would have provided. Nonetheless, decisions not subsequently retracted on the path to the solution can be considered better than those that were retracted, so we take them as positive examples. We also know that the roots of retracted subtrees (*regressions*) were errors, and therefore take them as negative examples. (Note that

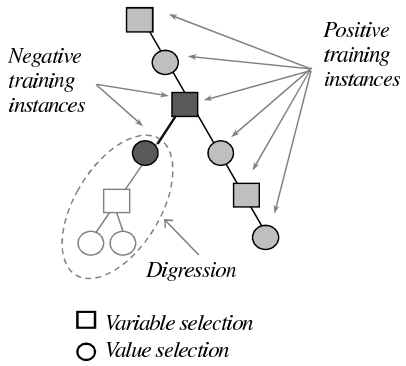

Fig. 5.3 The extraction of positive and negative training instances from the trace of a successful CSP search

assigning value v to variable X is not an error if in some solution of the problem $X = v$; what we address here is whether assigning v to X given the current partial instantiation is an error.)

As in Figure 5.3, *positive training instances* are those made along an error-free path extracted from a solution trace. *Negative training instances* are value selections that led to a digression, as well as variable selections whose subsequent value assignment failed. (Given correct value selections, any variable ordering can produce a backtrack-free solution; ACE deems a variable selection inadequate if the subsequent value assignment to that variable failed.) Decisions below the root of a digression do not become training instances.

Although machine learning assumes that training and testing examples come from the same population, a solver's experience in a class of CSPs will not be uniform, and thus the solver is likely to be misled. Despite the common characterization of the examples, the difficulty a solver has on problems in the same class has been shown to have a heavy tail, that is, to have a Pareto-normal distribution. This means that a portion of problems in the class will be very difficult for the solver's algorithm, and that portion will not decrease exponentially. Furthermore, although CSPs in the same class are ostensibly similar, there is evidence that their difficulty may vary substantially for a given search algorithm [23]. Thus our learner will inevitably be confronted with a non-uniform, heuristically selected set of training examples.

5.5.3 Performance Assessment

CSP search performance is traditionally gauged by the size of the search tree expanded (number of partial instantiations) and elapsed CPU time. Here, a solver executes in a *run* of two phases: a *learning phase* during which it attempts to solve a sequence of CSPs from a given class, and a *testing phase* during which it attempts to solve a sequence of hitherto unseen problems drawn from the same class. Because the class is of uneven difficulty, the problems in the learning phase may not be in-

dicative of the class as a whole. Thus we average the solver's performance over an *experiment*, here a set of ten runs.

Ideally, a solver should also consider its performance more broadly. It should be aware of its general progress on the class. If it believes it can learn no more, it should terminate learning itself and proceed to testing. And if it believes that learning is not going well, it should elect to discard what it has learned and start over. The solver constructed to meet these expectations is called ACE.

5.6 ACE

When *ACE* (the Adaptive Constraint Engine) learns to solve a class of binary CSPs, it customizes a weighted mixture of heuristics for the class [13]. ACE is based on *FORR*, an architecture for the development of expertise from multiple heuristics [12]. ACE's search algorithm (in Figure 5.4) 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.

5.6.1 Decision Hierarchy

Decision-making procedures in ACE are called *Advisors*. They are organized into three tiers, and presented with the current set of choices (variables or values). Tier-1 Advisors are correct and quick, and preordered by the user. If any of them approves a choice, it is executed. (For example, *Victory* recommends any value from the domain of the final unassigned variable. Since inference has already removed inconsistent values, any remaining value produces a solution.) Disapproval from any tier-1 Advisor eliminates some subset of choices; the remaining choices are passed to the next Advisor. (The set of choices is not permitted to go empty.) Tier-2 Advisors address subgoals; they are outside the scope of this chapter and not used in the experiments reported here.

The work described here focuses on the Advisors in tier 3. Each tier-3 Advisor *comments* upon (produces a strength for) some of its favored choices, those whose metric scores are among the f most favored. Because a metric can return identical values for different choices, an Advisor usually makes many more than f comments. (Here $f = 5$, unless otherwise stated.) The *strength s(A, c)* is the degree of support from Advisor A for choice c. Each tier-3 Advisor's view is based on a descriptive metric. All tier-3 Advisors are consulted together. As in Figure 5.4, a decision in tier 3 is made by *weighted voting*, where the strength $s(A, c)$ given to choice c by Advisor A is multiplied by the *weight* $w(A)$ of Advisor A. All weights are initialized to 0.05, and then learned for a class of problems by the processes described below. The discount factor $q(A)$ in (0,1] modulates the influence of Advisor A until it has commented often enough during learning. As data is observed on A, $q(A)$ moves

Fig. 5.4: Search in ACE with a weighted mixture of variable-ordering Advisors from A_{var}, and value-ordering Advisors fromA_{val}. *q(A)* is the discount factor. *w(A)* is the weight of Advisor *A*. *s(A, c)* is the strength of Advisor *A* for choice *c*.

Search (p, Avar , Aval)
Until problem p is solved or the allocated resources are exhausted
Select unvalued variable v

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

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

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

Update domains of all unvalued variables **inference**
Unless domains of all unvalued variables are nonempty
return to a previous alternative value **retraction**

toward 1, effectively increasing the impact of A on a given class as its learned weight becomes more trustworthy.

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

5.6.2 Weight Learning

Given a class of binary, solvable problems, ACE's goal is to formulate a mixture of Advisors whose joint decisions lead to effective search on a class of CSPs. Its learning scenario specifies that the learner seeks only one solution to one problem at a time, and learns only from problems that it solves. There is no information about whether a single different decision might have produced a far smaller search tree. This is therefore a form of incremental, self-supervised reinforcement learning based only on limited search experience and incomplete information. As a result, any weight-learning algorithm for ACE must select training examples from which to learn, determine what constitutes a heuristic's support for a decision, and specify a way to assign credits and penalties.

ACE learns weights only after it solves a problem. ACE's two most successful approaches to weight learning are Digression-based Weight Learning (*DWL*) [13] and Relative Support Weight Learning (*RSWL*) [35]. It uses them to update the weights of its tier-3 Advisors. Both weight-learning algorithms glean training instances from their own (likely imperfect) successful searches (as described in Section 5.5.2).

Fig. 5.5: Learning weights for Advisors. The *Search* algorithm is defined in Figure 5.4

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

A weight-learning algorithm defines what it means to support a decision. Under DWL, an Advisor is said to support only those decisions to which it assigned the highest strength. In contrast, RSWL considers all strengths. The *relative support* of an Advisor for a choice is the normalized difference between the strength the Advisor assigned to that choice and the average strength it assigned to all available choices at that decision point. For RSWL, an Advisor supports a choice if its relative support for that choice is positive, and opposes that choice if its relative support is negative. As in Figure 5.5, heuristics that support positive training instances receive credits, and heuristics that support negative training instances receive penalties. For both DWL and RSWL, an Advisor's weight is the averaged sum of the credits and penalties it receives, but the two weight-learning algorithms determine credits and penalties differently.

DWL reinforces Advisors' weights based on the size of the search tree and the size of each digression. An Advisor that supports a positive training instance is rewarded with a weight increment that depends upon the size of the search tree, relative to the minimal size of the search tree in all previous problems. An Advisor that supports a negative training instance is penalized in proportion to the number of search nodes in the resultant digression. Small search trees indicate a good variable order, so the variable-ordering Advisors that support positive training instances from a successful small tree are highly rewarded. For value ordering, however, a small search tree is interpreted as an indication that the problem was relatively easy (i.e., any value selection would likely have led to a solution), and therefore results in only small weight increments. In contrast, a successful but large search tree suggests that a problem was relatively difficult, so value-ordering Advisors that support positive training instances from it receive substantial weight increments [13].

RSWL is more local in nature. With each training instance RSWL reinforces weights based upon the distribution of each heuristic's preferences across all the available choices. RSWL reinforces weights based both upon relative support and upon an estimate of how difficult it is to make the correct decision. For example, an Advisor that strongly singles out the correct decision in a positive training instance

receives more credit than a less discriminating Advisor, and the penalty for a wrong choice from among a few is harsher than for a wrong choice from among many.

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

5.7 Techniques that Improve Learning

This section describes four techniques that use both search performance and problem difficulty to adapt learning: full restart, random subsets of heuristics, consideration of decision difficulty, and the nuances of preferences. Section 8 provides empirical demonstrations of their efficacy.

5.7.1 Full Restart

From a large initial list of heuristics that contains minimizing and maximizing versions of many metrics, some perform poorly on a particular class of problems (*class-inappropriate heuristics*) while others perform well (*class-appropriate heuristics*). In some cases, class-inappropriate heuristics occasionally acquire high weights on an initial problem and then control subsequent decisions. As a result, subsequent problems may have extremely large search trees.

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

Under *full restart*, however, ACE monitors the frequency and the order of unsolved problems in the problem sequence. If it deems the current learning attempt not promising, ACE abandons the learning process (and any learned weights) and begins learning on new problems with freshly initialized weights [34]. Note that full restart is different from restart on an individual problem, discussed in Section 5.3, which diversifies the search for a solution to that problem alone [20]. We focus here only on the impact of full restart of the entire learning process.

The node limit is a critical parameter for full restart. Because ACE abandons a problem if it does not find a solution within the node limit, the node limit is the criterion for unsuccessful search. Since the full restart threshold directly depends upon the number of failures, the node limit is the performance standard for full restart. The node limit also controls resources; lengthy searches permitted under high node limits are expensive.

Resource limits and full restart impact the cost of learning in complex ways. With higher node limits, weights can eventually recover without the use of full restart, but recovery is more expensive. With lower node limits, the cost of learning (total number of nodes across all learning problems) with full restart is slightly higher than without it. The learner fails on all the difficult problems, and even on some of medium difficulty, repeatedly triggering full restart until the *weight profile* (the set of tier-3 weights) is good enough to solve almost all the problems. Full restart abandons some problems and uses additional problems, which increases the cost of learning. The difference in cost is small, however, since each problem's cost is subject to a relatively low node limit. As the node limit increases, full restart produces fewer inadequate runs, but at a higher cost. It takes longer to trigger full restart because the learned weight profile is deemed good enough and failures are less frequent. Moreover, with a high node limit, every failure is expensive. When full restart eventually triggers, the prospect of relatively extensive effort on further problems is gone. Because it detects and eliminates unpromising learning runs early, full restart avoids many costly searches and drastically reduces overall learning cost. Experimental results and further discussion of full restart appear in Section 5.8.1.

5.7.2 Learning with Random Subsets

The interaction among heuristics can also serve as a filter during learning. Given a large and inconsistent initial set of heuristics, many class-inappropriate ones may combine to make bad decisions, and thereby make it difficult to solve any problem within a given node limit. Because only solved problems provide training instances for weight learning, no learning can take place until some problem is solved. Rather than consult all its Advisors at once, ACE can randomly select a new subset of Advisors for each problem, consult them, make decisions based on their comments, and update only their weights [37]. This method, *learning with random subsets*, eventually uses a subset in which class-appropriate heuristics predominate and agree on choices that solve a problem.

It is possible to preprocess individual heuristics on representatives of a problem class (possibly by racing [4]), and then eliminate from the candidate heuristics those with poor individual performance. That approach, however, requires multiple solution attempts on some set of problems. Multiple individual runs would consume more computational resources, because many different Advisors reference (and share the values of) the same metrics in our approach. Moreover, elimination of poorly performing heuristics after preprocessing might prevent the discovery of

important synergies (e.g., tiebreakers) between eliminated heuristics and retained ones.

For a fixed node limit and set of heuristics, an underlying assumption here is that the ratio of class-appropriate to class-inappropriate heuristics determines whether a problem is likely to be solved. When class-inappropriate heuristics predominate in a set of heuristics, the problem is unlikely to be solved and no learning occurs. The selection of a new random subset of heuristics for each new problem, however, should eventually produce some subset S with a majority of class-appropriate heuristics that solves its problem within a reasonable resource limit. As a result, the Advisors in S will have their weights adjusted. On the next problem, the new random subset S' is likely to contain some low-weight Advisors outside of S, and some reselected from S. Any previously successful Advisors from S that are selected for S' will have larger positive weights than the other Advisors in S', and will therefore heavily influence search decisions. If S succeeded because it contained more class-appropriate than class-inappropriate heuristics, $S \cap S'$ is also likely to have more class-appropriate heuristics and therefore solve the new problem, so again those that participate in correct decisions will be rewarded. On the other hand, in the less likely case that the majority of $S \cap S'$ consists of reinforced, class-inappropriate heuristics, the problem will likely go unsolved, and the class-inappropriate heuristics will not be rewarded further.

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

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

$$p = \sum_{k=\left\lfloor \frac{r}{2}+1 \right\rfloor}^{r} \frac{\binom{a}{k}\binom{b}{r-k}}{\binom{a+b}{r}} \tag{5.1}$$

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

$$p = \sum_{i=1}^{\infty} i(1-p)^{i-1}p = \frac{1}{p}. \tag{5.2}$$

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 $r = 4$, equation (1) evaluates to 0.14 and equation (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.

Weight convergence is linked to subset size. Weights converge faster when subsets are larger. When the random subsets are smaller, subsequent random subsets are less likely to overlap with those that preceded them, and therefore less likely to include Advisors whose weights have been revised. As a result, failures occur often, even after some class-appropriate heuristics receive high weights. ACE monitors its learning progress, and can adapt the size of random subsets. In this scenario, random subsets are initially small, but as learning progresses and more Advisors participate and obtain weights, the size of the random subsets increase. This makes overlap more likely, and thereby speeds learning. Experimental results and further discussion of learning with random subsets appear in Section 5.8.2.

5.7.3 Learning Based on Decision Difficulty

Correct easy decisions are less significant for learning; it is correct difficult decisions that are noteworthy. Thus it may be constructive to estimate the difficulty of each decision the solver faces as if it were a fresh problem, and adjust Advisors' weights accordingly. Our rationale for this is that, on easy problems, any decision leads to a solution. Credit for an easy decision effectively increases the weight of Advisors that support it, but if the decision was made during search, those Advisors probably already had high weights. ACE addresses this issue with two algorithms, each dependent upon a single parameter: RSWL-κ and RSWL-d.

Constrainedness, as measured by κ, has traditionally been used to identify hard classes of CSPs [18]. κ depends upon *n, d, m,* and *t,* as defined in Section 3:

$$\kappa = \frac{n-1}{2} \cdot d \cdot \log_m \frac{1}{1-t}. \tag{5.3}$$

For every search algorithm, and for fixed n and m, hard problem classes have κ close to 1. *RSWL-*κ uses equation (3) to measure the difficulty κ_P of subproblem P at each decision point. For a given parameter k, RSWL-κ gives credit to an Advisor only when it supports a positive training instance derived from a search state where $|\kappa_P - 1| < k$. RSWL-κ penalizes an Advisor only when it supports a negative training instance derived from a search state where $|\kappa_P - 1| > k$. The calculation of κ_P on every training instance is computationally expensive.

RSWL-d uses the number of unassigned variables at the current search node as a rough, quick estimate of problem hardness. Decisions at the top of the search tree are known to be more difficult [38]. For a given parameter h, no penalty is given at all for any decision in the top h percent of the nodes in the search tree, and no credit

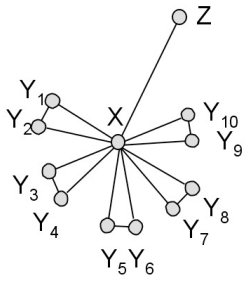

Fig. 5.6 A constraint graph for a CSP problem on 12 variables

is given for any decision below them. Experimental results and further discussion of learning with decision difficulty appear in Section 5.8.3.

5.7.4 Combining Heuristics' Preferences

The preferences expressed by heuristics can be used to make decisions during search. The intuition here is that comparative nuances, as expressed by preferences, contain more information than just what is "best." Recall that each heuristic reflects an underlying metric that returns a *score* for each possible choice. Comparative opinions (here, *heuristics' preferences*) can be exploited in a variety of ways that consider both the scores returned by the metrics on which these heuristics rely and the distributions of those scores across a set of possible choices.

The simplest way to combine heuristics' preferences is to scale them into some common range. Mere ranking of these scores, however, reflects only the preferences for one choice over another, not the extent to which one choice is preferred over another. For example in Figure 5.6, the degrees of variables X and Y_1 differ by 9, while the degrees of Y_1 and Z differ by only 1. Nonetheless, ranking by degree assigns equally spaced strengths (3, 2 and 1, respectively) to X, Y_1, and Z. Ranking also ignores how many choices share the same score. For example, in Table 5.4, the ranks of choices Y_1 and Z differ by only 1, although the heuristic prefers only one choice over Y_1 and 11 choices over Z. We have explored several methods that express Advisors' preferences and address those shortcomings [36].

Linear interpolation not only considers the relative position of scores, but also the actual differences between them. Under linear interpolation, strength differences are proportional to score differences. For example, in Table 5.4, strengths can be determined by the value of the linear function through the points (11, 3) and (1, 1). Instead of strength 2 for all the Y variables, linear interpolation gives them strength 1.2, which is closer to the strength 1 given to variable Z, because the degrees of the Y variables are closer to the degree of Z. The significantly higher degree of variable X is reflected in the distance between its strength and those given to the other variables.

Table 5.4: The impact of different preference expression methods on a single metric

Variables	***X***	$\boldsymbol{Y_1, Y_2, ..., Y_{10}}$	***Z***
Degree metric scores	11	2	1
Rank strength	3	2	1
Linear strength	3.0	1.2	1.0
Borda-w strength	11.0	1.0	0.0
Borda-wt strength	12.0	11.0	1.0

The *Borda methods* were inspired by an election method devised by Jean-Charles de Borda in the late eighteenth century [39]. Borda methods consider the total number of available choices, the number of choices with a smaller score and the number of choices with an equal score. Thus the strength for a choice is based on its position relative to the other choices.

The first Borda method, *Borda-w*, awards a point for each *win* (metric score higher than the score of some other commented choice). Examples for *Borda-w* strengths are also shown in Table 5.4. The set of lowest-scoring variables (here, only *Z*) always has strength 0. Because every *Y* variable outscored only *Z*, the strength of any *Y* variable is 1. The highest-scoring choice *X* outscored 11 choices, so *X*'s strength is 11.

The second Borda method, *Borda-wt,* awards one point for each win and one point for each *tie* (score equal to the score of some other choice). It can be interpreted as emphasizing losses. The highest-scoring set of variables (here, only *X*) always has strength equal to the number of variables to be scored. For example, in Table 5.4, no choice outscored the highest-scoring choice *X,* so its strength is 12, one choice (*X*) outscored the *Y* variables, so their strengths are reduced by one point (12 – 1 = 11), and 11 choices outscored *Z*, resulting in strength 1.

The difference between the two Borda methods is evident when many choices share the same score. *Borda-w* considers only how many choices score lower, so that a large subset results in a big gap in strength between that subset and the previous (more preferred) one. Under *Borda-wt*, a large subset results in a big gap in strength between that subset and the next (less preferred) one. In Table 5.4, for example, the 10 *Y* variables share the same score. Under *Borda-w*, the difference between the strength of every *Y* variable and *X* is 10, while the difference between the strength of any *Y* variable and *Z* is only 1. Under *Borda-wt*, however, the difference between the strength of any *Y* and *X* is only 1, while the difference between the strength of any *Y* and *Z* is 10. The Borda approach can be further emphasized by making a point inversely proportional to the number of subsets of tied values. Experimental results and further discussion of learning with preferences using this emphasis appear in Section 5.8.4.

5.8 Results

The methods in the previous section are investigated here with ACE. As described in Section 5.5.3, each run under ACE is a learning phase followed by a testing phase. During a learning phase ACE refines its Advisor weights; the testing phase provides a sequence of fresh problems with learning turned off. All of the following experiments average results across ten runs, and differences cited are statistically significant at the 95% confidence level.

In the experiments that follow, learning terminated after 30 problems, counting from the first solved problem, or terminated if no problem in the first 30 was solved at all. Under full restart, more learning problems can be used, but the upper bound for the total number of problems in a learning phase was always 80. For each problem class, every testing phase used the same 50 problems. When any ten of the 50 testing problems went unsolved within the node limit, learning in that run was declared *inadequate* and further testing was halted. In every learning phase, ACE had access to 40 tier-3 Advisors, 28 for variable selection and 12 for value selection (described in the Appendix). During a testing phase, ACE used only those Advisors whose learned weights exceeded those of their respective benchmarks.

5.8.1 Full Restart Improves Performance

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

Under every node limit tested, full restart produced more runs that were successful, as Figure 5.7 illustrates. At lower node limits, Figure 5.8 shows that better testing performance came with a learning cost similar to or slightly higher than the cost without full restart. At higher node limits, the learning cost was considerably lower with full restart. With very low node limits (200 or 300 nodes), even with full restart DWL could not solve all the problems. During learning, many problems went unsolved under a low node limit and therefore did not provide training in-

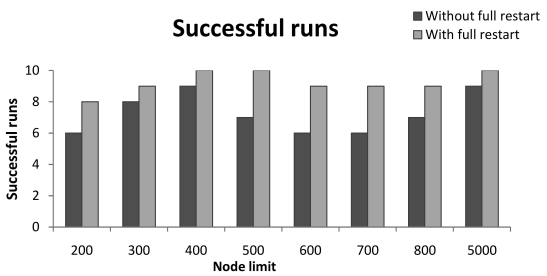

Fig. 5.7 Number of successful runs (out of ten) on <30, 8, 0.26, 0.34> problems under different node limits

stances. On some (inadequate) runs, no solution was found to any of the first 30 problems, so learning was terminated without any weight changes. When the node limit was somewhat higher (400 nodes), more problems were solved, more training instances became available and more runs were successful. These reasonably low node limits set a high standard for the learner; only weight profiles well tuned to the class will solve problems within them and thereby provide good training instances. Further increases in the node limit (500, 600, 700 and 800 nodes), however, did not further increase the number of successful runs. Under higher node limits, problems were solved even with weight profiles that were not particularly good for the class, and may have produced training instances that were not appropriate. Under extremely high node limits (5,000 nodes), problems were solved even under inadequate weight profiles, but the weight-learning mechanism was able to recover a good weight profile, and again the number of successful runs increased.

Similar performance was observed under RSWL and on *Geo* and the other model B classes identified in Section 5.3, but not on the *Comp* problems. Some *Comp* problems go unsolved under any node limit, while many others are solved. Because failures there are sporadic, they do not trigger full restart. The use of full restart on them does not improve learning, but it does not harm it either. (Data omitted.) Full restart is therefore used throughout the remainder of these experiments.

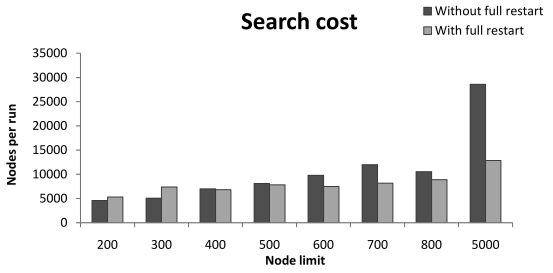

Fig. 5.8 Learning cost, measured by the average number of nodes per run, on <30, 8, 0.26, 0.34> problems under different node limits

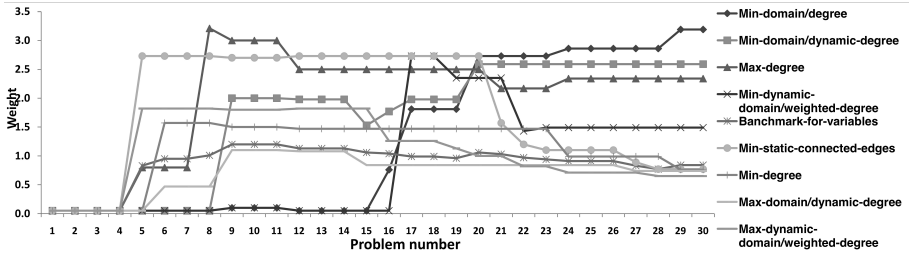

Fig. 5.9: Weights of eight variable-ordering Advisors and their common benchmark during learning after each of 30 problems in <50, 10, 0.38, 0.2> on a single run

5.8.2 Random Subsets Improve Performance

When the full complement of Advisors was present but resource limits were strict, even with full restart ACE sometimes failed to solve problems in some difficult classes. Random subsets corrected this. Figure 5.9 illustrates weight convergence during learning with random subsets. On this run some Advisors (e.g., *Min-domain/dynamic-degree* and *Min-static-degree*) recovered from initially inadequate weights. The figure tracks the weights of eight variable-ordering heuristics and their common benchmark (described in Section 5.6.2) after each of 30 problems. Here the problems were drawn from <50, 10, 0.38, 0.2>, and 30% of the variable-ordering Advisors and 30% of the value-ordering Advisors were randomly selected for each problem. Plateaus in weights correspond to problems where the particular heuristic was not selected for the current random subset, or the problem went unsolved, so that no learning or weight changes occurred. The first four problems were *early failures* (problems that went unsolved under initial weights, before any learning occurred). When the fifth problem was solved, some class-inappropriate Advisors received high weights from its training instances. On the next few problems, either highly weighted but class-inappropriate heuristics were reselected and the problem went unsolved and no weights changed, or some class-appropriate Advisors were selected and gained high weights. Eventually the latter began to dominate decisions, so that the disagreeing class-inappropriate Advisors had their weights reduced. After the 21st problem, when the weight of *Min-static-connected-edges* had significantly decreased, the weights clearly separated the class-appropriate Advisors from the class-inappropriate ones. Afterwards, as learning progressed, the weights stabilized.

Experiments that illustrate the benefits of random subsets tested four ways to choose the Advisors from each problem:

1. *All* used all the Advisors on every problem.
2. *Fixed* chose a fixed percentage q (30% or 70%), and then chose q variable-ordering Advisors and q value-ordering Advisors, without replacement.

3. *Varying* chose a random value r in [30, 70] for each problem, and then chose r percent of the variable-ordering Advisors and r percent of the value-ordering Advisors, without replacement.
4. *Incremental* initially selected q of the variable-ordering Advisors and q of the value-ordering Advisors. Then, for each subsequent problem, it increased the sizes of the random subsets in proportion to the number of Advisors whose weight was greater than their initially assigned weight.

On problems in <50, 10, 0.18, 0.37>, Table 5.5 compares these approaches for learning with random subsets of Advisors to learning with all the Advisors at once. When all 40 Advisors were consulted, the predominance of class-inappropriate Advisors sometimes prevented the solution of any problem under the given node limit, so that some learning phases were terminated after 30 unsolved problems. In those runs no learning occurred. With random subsets, however, adequate weights were learned on every run, and there were fewer early failures.

Table 5.5: Early failures, successful runs, decision time and percentage of computation time during learning with random subsets of Advisors, compared to computation time with all the Advisors on problems in <50, 10, 0.18, 0.37>

Advisors	***Early failures per run***	***Successful runs***	***Time per learning decision***	***Learning time per run***
All	27.0	4	100.00%	100.00%
Fixed q = 70%	5.2	10	74.30%	24.39%
Varying $r \in$ [30%,70%]	1.9	10	65.84%	20.74%
Incremental q = 30%	1.8	10	75.35%	19.76%
Fixed q = 30%	3.1	10	55.39%	21.85%

Table 5.5 also demonstrates that using random subsets significantly reduces learning time. The time to select a variable or a value is not necessarily directly proportional to the number of selected Advisors. This is primarily because dual pairs of Advisors share the same fundamental computational cost: calculating their common metric. For example, the bulk of the work for *Min-product-domain-value* lies in the one-step lookahead that calculates (and stores) 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 that are not their duals. The reduction in total computation time per run also reflects any reduction in the number of learning problems.

The robustness of learning with random subsets is demonstrated with experiments documented in Table 5.6 that begin with fewer Advisors, a majority of which are class-inappropriate. Based on weights from successful runs with all Advisors, Advisors were first identified as class-appropriate or class-inappropriate for problems in <50, 10, 0.18, 0.37>. ACE was then provided with two different sets A_{var}

of variable-ordering Advisors in which class-inappropriate Advisors outnumbered class-appropriate ones (nine to six or nine to four). When all the provided Advisors were consulted, the predominance of class-inappropriate Advisors effectively prevented the solution of any problem under the given node limit and no learning took place. When learning with random subsets, as the size of the random subsets decreased, the number of successful runs increased. As a result, random subsets with fixed $q = 30\%$ is used throughout the remainder of these experiments.

Table 5.6: Learning with more class-inappropriate than class-appropriate Advisors on problems in <50, 10, 0.18, 0.37>. Smaller, fixed-size random subsets appear to perform best

	6 class-appropriate 9 class-inappropriate		**4 class-appropriate 9 class-inappropriate**	
Advisors	***Early failures***	***Successful runs***	***Early failures***	***Successful runs***
All	30.0	0	30.0	0
Fixed $q = 70\%$	21.2	7	30.0	0
Varying $r \in [30\%, 70\%]$	8.4	10	17.6	6
Incremental $q = 30\%$	3.8	10	12.0	9
Fixed $q = 30\%$	5.1	10	13.3	10

5.8.3 The Impact of Decision Difficulty

Consideration of relative support and some assessment of problem difficulty can improve testing performance, as shown in Table 5.7. On problems in <50, 10, 0.38, 0.2>, RSWL solved more problems during both learning and testing than did DWL, and required fewer full restarts. Moreover, both RSWL-κ and RSWL-d solved more problems with fewer nodes during testing.

5.8.4 Performance with Preferences

We tested both linear interpolation and the Borda methods on the CSP classes identified in Section 5.3. On the unstructured model B problems, preference expression made no significant difference. On *Comp*, however, across a broad range of node limits, preference expression had an effect, as shown in Table 5.8.

Initially, most variables in the central component score similarly on most metrics, and most variables in the satellite score similarly to one another but differently from

Table 5.7: Learning and testing performance with different preference expression methods on <50, 10, 0.38, 0.2> problems. Bold figures indicate statistically significant reductions, at the 95% confidence level, in the number of nodes and in the percentage of solved problems compared to search under DWL

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

Table 5.8: Testing performance with RSWL on *Comp* problems with reduced node limits and a variety of preference expression methods. Although there appear to be substantial differences, the variance is such that only the figure in bold is a statistically significant improvement at the 95% confidence level

Node	Ranking		Borda-w		Borda-wt		Linear	
Limit	Nodes	Solved	Nodes	Solved	Nodes	Solved	Nodes	Solved
5000	161.1	97.8%	134.1	98.0%	638.5	88.6%	164.2	97.8%
1000	161.1	97.8%	134.1	98.0%	564.7	89.8%	164.2	97.8%
500	161.5	97.8%	121.1	98.2%	728.5	86.6%	164.2	97.8%
100	161.4	97.8%	111.6	98.4%	642.1	88.2%	163.7	97.8%
35	160.4	97.8%	111.7	98.4%	660.0	89.0%	**33.5**	100.0%

those in the central component. Under *Borda–wt*, if only a few choices score higher, the strength of the choices from the next lower-scoring subset is close enough to influence the decision. If there are many high-scoring choices, in the enhanced version the next lower subset will have a much lower strength, which decreases its influence. Moreover, when many choices share the same score, they are penalized for failure to discriminate, and their strength is lowered. When *Borda–w* assigns lower strengths to large subsets from the central component, it makes them less attractive. That encourages variables in the satellites to be selected first; this is often the right way to solve such problems.

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

5.9 Conclusions and Future Work

Our fundamental underlying assumption is that the right way to make search decisions in a class of CSPs has some uniformity, that is, that it is possible to learn from one problem how to solve another. ACE tries to solve a CSP from a class, and, if it finds a solution, it extracts training examples from the search trace. If it fails to solve the CSP, however, given the size of the search space, the solver will learn nothing from the effort it expended.

Given a training example, the learning algorithms described here reinforce heuristics that prove successful on a set of problems and discard those that do not. Our program represents its learned knowledge about how to solve problems as a weighted sum of the output from some subset of its heuristics. Thus the learner's task is both to choose the best heuristics and to weight them appropriately.

ACE is a successful, adaptive solver. It learns to select a weighted mixture of heuristics for a given problem class, one that produces search trees smaller than those from outstanding individual heuristics in the CSP literature. ACE learns from its own search performance, based upon the accuracy, intensity, frequency and distribution of its heuristics' preferences. ACE adapts its decision making, its reinforcement policy, and its heuristic selection mechanisms effectively.

Our current work extends these ideas on several fronts. Under an option called *Pusher*, ACE consults the single highest-weighted tier-3 variable-ordering heuristic below the maximum search depth at which it has experienced backtracking on other problems in the same class [13]. Current work includes learning different weight profiles for different stages in solving a problem, where stages are determined by search tree depth or the constrainedness of the subproblem at the decision point. A generalization of that approach would associate weight profile(s) with an entire benchmark family of problems, and begin with the weights of the most similar benchmark family for each new problem instance.

Rather than rely on an endless set of fresh problems, we plan to reuse unsolved problems and implement boosting with little additional effort during learning [42]. A major focus is the automated selection of good parameter settings for an individual class (including the node limit and full-restart parameters), given the results in [24]. We also intend to extend our research to classes containing both solvable and unsolvable problems, and to optimization problems. Finally, we plan to study this approach further in light of the factor analysis evidence of strong correlations between CSP ordering heuristics [47]. Meanwhile, ACE proficiently tailors a mixture of search heuristics for each new problem class it encounters.

Appendix

Two vertices with an edge between them are *neighbors*. Here, the *degree of an edge* is the sum of the degrees of its endpoints, and the *edge degree of a variable* is the

sum of the edge degrees of the edges on which it is incident. Each of the following metrics produces two Advisors.

Metrics for variable selection were static degree, dynamic domain size, FF2 [43], 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 [6], and ratio of dynamic domain size to weighted degree.

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.

Acknowledgements ACE is a joint project with Eugene Freuder and Richard Wallace of The Cork Constraint Computation Centre. Dr. Wallace made important contributions to the DWL algorithm. This work was supported in part by the National Science Foundation under IIS-0328743, IIS-0739122, and IIS-0811437.

References

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

[2] Ali, K., Pazzani, M. (1996). Error reduction through learning multiple descriptions. Machine Learning, 24, pp. 173-202.

[3] Arbelaez, A., Hamadi, Y., Sebag, M. (2009). Online Heuristic Selection in Constraint Programming. International Symposium on Combinatorial Search (SoCS-2009), Lake Arrowhead, CA.

[4] Mauro Birattari, Thomas Stützle, Luis Paquete, Klaus Varrentrapp: A Racing Algorithm for Configuring Metaheuristics. Genetic and Evolutionary Computation Conference (GECCO-2002), New York, NY, pp. 11-18.

[5] Borrett, J. E., Tsang, E. P. K., Walsh, N. R. (1996). Adaptive Constraint Satisfaction: The Quickest First Principle. 12th European Conference on AI (ECAI-1996), Budapest, Hungary, pp. 160-164.

[6] Boussemart, F., Hemery, F., Lecoutre, C., Sais, L. (2004). Boosting Systematic Search by Weighting Constraints. 16th European Conference on AI (ECAI-2004), Valencia, Spain, pp. 146-150.

[7] Burke, E., Hart, E., Kendall, G., Newall, J., Ross, P., Schulenburg, S. (2003). Hyperheuristics: An emerging direction in modern research technology. Handbook of Metaheuristics, Dordrecht: Kluwer Academic Publishers, pp. 457-474.

[8] Chakhlevitch, K., Cowling, P. (2008). Hyperheuristics: Recent Developments. C. Cotta, M. Sevaux, K. Sorensen (Eds.), Adaptive and Multilevel Metaheuristics, Studies in Computational Intelligence, Springer, pp. 3-29 .

[9] Cicirello, V. A., Smith, S. F. (2004). Heuristic Selection for Stochastic Search Optimization: Modeling Solution Quality by Extreme Value Theory. Principles and Practice of Constraint Programming (CP-2004), Toronto, Canada, LNCS 3258, pp. 197-211.

[10] D'Andrade, R. G. (1990). Some Propositions About the Relations Between Culture and Human Cognition. J. W. Stigler, R. A. Shweder, G. Herdt (Eds.), Cultural Psychology: Essays on Comparative Human Development, Cambridge University Press, Cambridge 1990, pp. 65-129.

[11] Dietterich, T. G. (2000). Ensemble methods in machine learning. J. Kittler, F. Roli (Eds.) First International Workshop Multiple Classifier Systems, MCS-2000, Cagliari, Italy, LNCS 1857, pp. 1-15.

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

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

[14] Freund, Y., Schapire, R. (1996). Experiments with a new boosting algorithm. Thirteenth International Conference on Machine Learning (ICML-96), Bari, Italy, pp. 148-156.

[15] Fukunaga, A. S. (2002). Automated Discovery of Composite SAT Variable-Selection Heuristics. Eighteenth National Conference on Artificial Intelligence (AAAI-2002), Edmonton, Canada, pp. 641-648.

[16] Gagliolo, M., Schmidhuber, J. (2006). Dynamic Algorithm Portfolios. Ninth International Symposium on Artificial Intelligence and Mathematics, Special Issue of the Annals of Mathematics and Artificial Intelligence, 47(3-4), pp. 295-328.

[17] Gebruers, C., Hnich, B., Bridge, D., Freuder, E. (2005). Using CBR to Select Solution Strategies in Constraint Programming. Case-Based Reasoning Research and Development. LNCS 3620, pp. 222-236.

[18] Gent, I. P., Prosser, P., Walsh, T. (1996). The Constrainedness of Search. The Thirteenth National Conference on Artificial Intelligence (AAAI-1996), Portland, Oregon, pp. 246-252.

[19] Gomes, C., Fernandez, C., Selman, B., Bessière, C. (2004). Statistical Regimes Across Constrainedness Regions. Principles and Practice of Constraint Programming (CP-2004), Toronto, Canada, LNCS 3258, pp. 32-46.

[20] Carla P. Gomes, Bart Selman, Henry A. Kautz: Boosting Combinatorial Search Through Randomization, The Fifteenth National Conference on Artificial Intelligence (AAAI-1998), Madison, Wisconsin, pp. 431-437.

[21] Gomes, C. P., Selman, B. (2001). Algorithm portfolios. Artificial Intelligence, 126(1-2), pp. 43-62.

[22] Hansen, L., Salamon, P. (1990). Neural network ensembles. IEEE Transactions on Pattern Analysis and Machine Intelligence, 12, pp. 993-1001.

[23] Hulubei, T., O'Sullivan, B. (2005). Search Heuristics and Heavy-Tailed Behavior. Principles and Practice of Constraint Programming (CP-2005), Barcelona, Spain, pp. 328-342.

[24] Hutter, F., Hamadi, Y., Hoos, H. H., Leyton-Brown, K. (2006). Performance Prediction and Automated Tuning of Randomized and Parametric Algorithms. Principles and Practice of Constraint Programming (CP-2006), Nantes, France, pp. 213-228.

[25] Johnson, D. S., Aragon, C. R., McGeoch, L. A., Schevon, C. (1989). Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning. Operations Research, 37(6), 865-892.

[26] Kivinen, J., Warmuth, M. K. (1999). Averaging expert predictions. Computational Learning Theory: 4th European Conference (EuroCOLT-1999), Nordkirchen, Germany, pp. 153-167.

[27] Lecoutre, C., Boussemart, F., Hemery, F. (2004). Backjump-based techniques versus conflict directed heuristics. 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI-2004), Boca Raton, Florida, USA, pp. 549-557.

[28] Minton, S., Allen, J. A., Wolfe, S., Philpot, A. (1995). An Overview of Learning in the Multi-TAC System. First International Joint Workshop on Artificial Intelligence and Operations Research, Timberline, Oregon, USA,

[29] Nareyek, A. (2004). Choosing Search Heuristics by Non-Stationary Reinforcement Learning. Metaheuristics: Computer Decision-Making, Kluwer Academic Publishers, Norwell, MA, USA, pp. 523-544.

[30] O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B. (2008). Using Case-based Reasoning in an Algorithm Portfolio for Constraint Solving. 19th Irish Conference on Artificial Intelligence and Cognitive Science (AICS-2008), Cork, Ireland.

[31] Opitz, D., Shavlik, J. (1996). Generating accurate and diverse members of a neural-network ensemble. Advances in Neural Information Processing Systems, 8, pp. 535-541.

[32] Otten, L., Gronkvist, M., Dubhashi, D. P. (2006). Randomization in Constraint Programming for Airline Planning. Principles and Practice of Constraint Programming (CP-2006), Nantes, France, pp. 406-420.

[33] Petrie, K. E., Smith, B. M. (2003). Symmetry breaking in graceful graphs. Principles and Practice of Constraint Programming (CP-2003), Kinsale, Ireland, LNCS 2833, pp. 930-934.

[34] Petrovic, S., Epstein, S. L. (2006). Full Restart Speeds Learning. 19th International FLAIRS Conference (FLAIRS-06), Melbourne Beach, Florida.

[35] Petrovic, S., Epstein, S. L. (2006). Relative Support Weight Learning for Constraint Solving. Workshop on Learning for Search, AAAI-06, Boston, Massachusetts, USA, pp. 115-122.

[36] Petrovic, S., Epstein, S. L. (2007). Preferences Improve Learning to Solve Constraint Problems. Workshop on Preference Handling for Artificial Intelligence (AAAI-07), Vancouver, Canada, pp. 71-78.

[37] Petrovic, S., Epstein, S. L. (2008). Random Subsets Support Learning a Mixture of Heuristics. International Journal on Artificial Intelligence Tools (IJAIT), 17(3), pp. 501-520.

[38] Ruan, Y., Kautz, H., Horvitz, E. (2004). The backdoor key: A path to understanding problem hardness. Nineteenth National Conference on Artificial Intelligence (AAAI-2004), San Jose, CA, USA, pp. 124-130.

[39] Saari, D. G. (1994). Geometry of voting. Studies in Economic Theory, Vol 3, Springer.

[40] Sabin, D., Freuder, E. C. (1997). Understanding and Improving the MAC Algorithm. Principles and Practice of Constraint Programming (CP-1997), Linz, Austria, LNCS 1330, pp. 167-181.

[41] Samulowitz, H., Memisevic, R. (2007). Learning To Solve QBF. 22nd National Conference on Artificial intelligence (AAAI-2007), Vancouver, Canada, pp. 255-260.

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

[43] Smith, B., Grant, S. (1998). Trying Harder to Fail First. European Conference on Artificial Intelligence (ECAI-1998), pp. 249-253.

[44] Streeter, M., Golovin, D., Smith, S. F. (2007). Combining multiple heuristics online. 22nd National Conference on Artificial Intelligence (AAAI-07), Vancouver, Canada, pp. 1197-1203.

[45] Terashima-Marin, H., Ortiz-Bayliss, J., Ross, P., Valenzuela-Rendón, M. (2008). Using Hyper-heuristics for the Dynamic Variable Ordering in Binary Constraint Satisfaction Problems, MICAI 2008, Advances in Artificial Intelligence, LNCS 5317, pp. 407-417.

[46] Valentini, G., Masulli, F. (2002). Ensembles of learning machines. Neural Nets (WIRN-2002), Vietri sul Mare, Italy, LNCS 2486, pp. 3-22.

[47] Wallace, R. J. (2005). Factor analytic studies of CSP heuristics. Principles and Practice of Constraint Programming (CP-2005), Barcelona, Spain, LNCS 3709, pp. 712-726.

[48] Whitley, D. (1989). The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. International Conference on Genetic Algorithms (ICGA-1989), pp. 116-121.

[49] Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K. (2007). SATzilla-07: The Design and Analysis of an Algorithm Portfolio for SAT. Principles and Practice of Constraint Programming (CP-2007), Providence, RI, USA, pp. 712-727.

[50] Young, H. P. (1988). Condorcet's theory of voting. The American Political Science Review, 82(4), 1231-1244.