The Adaptive Constraint Engine

Susan L. Epstein[1], Eugene C. Freuder[2], Richard Wallace[2], Anton Morozov[1]and Bruce Samuels[1]

[1] Department of Computer Science, Hunter College and The Graduate School of The City University of New York, New York, NY 10021, USA
susan.epstein@hunter.cuny.edu

[2] Cork Constraint Computation Centre, University College Cork, Cork, Ireland
e.freuder@4c.ucc.ie, rwallaceo@4c.ucc.ie

Abstract. The Adaptive Constraint Engine (ACE) seeks to automate the application of constraint programming expertise and the extraction of domain-specific expertise. Under the aegis of FORR, an architecture for learning and problem-solving, ACE learns search-order heuristics from problem solving experience. This paper describes ACE's approach, as well as new experimental results on specific problem classes. ACE is both a test-bed for CSP research and a discovery environment for new algorithms.

1. Introduction

The Adaptive Constraint Engine (*ACE*) is a program that learns from experience to be a better constraint programmer. A constraint satisfaction problem (*CSP*) involves a set of variables, a *domain* of values for each variable, and a set of *constraints* that specify which combinations of values are allowed. A *solution* to the problem assigns a value to each variable that satisfies all the constraints. Every (binary) CSP has an underlying *constraint graph*, where each variable is represented by a vertex whose possible labels are its domain values. An edge in the constraint graph appears between two vertices whenever there are constraints on the values of their corresponding vertices. One may think of an edge as labeled by the permissible pairs of values between its endpoints. The *degree* of a variable is the number of edges to it in the underlying constraint graph.

Currently, ACE learns search-order heuristics. We supply ACE with a set of primitive methods that characterize variables in the constraint graph, such as maximum domain size, minimum domain size, maximum degree, and minimum degree. These primitives are embedded in procedures called *Advisors* that collaborate on search-order decisions. For example, one Advisor, might recommend "choose the variable with maximum domain size" while another recommends "choose the variable with minimum domain size." As it solves problems, ACE learns which Advisors to value, and how to weight their advice.

ACE relies upon an underlying Advisor-based architecture called *FORR* (FOr the Right Reasons), which has proved successful in other domains [1-4]. In an earlier paper, some of us used FORR to construct GC, a graph-coloring program [5]. Here we extend the approach to general CSP's and incorporate the following advances:

- ACE can combine primitive Advisors to learn new Advisors.
- ACE demonstrates how learning on simple training instances can transfer to difficult test problems.
- ACE can learn different heuristics for different search stages (early, middle, late).

· ACE employs a new weight-learning algorithm.

ACE has demonstrated that it can not only rediscover classic heuristics, but also make new discoveries of equal or greater value. Specifically, ACE has discovered that maximizing the product of degree and forward-degree only in the early stage of search, and simply minimizing domain size thereafter, can be more effective in reducing the size of the search tree than minimizing the ratio of domain size to degree throughout the search. We also demonstrate that lessons learned by ACE can be applied in a more conventional context. In general, ACE provides a powerful testbed for exploring the nature of the search process with much greater ease and subtlety than has been heretofore available.

The next section of this paper provides an overview of FORR. Subsequent sections describe ACE, the experimental design and results, and our ability to transfer ACE's discoveries to a more traditional context. The final discussion includes an analysis of our results and plans for future work.

2. FORR

FORR is a problem-solving and learning architecture for the development of expertise from multiple heuristics. To make a decision, FORR combines the output of a set of procedures called Advisors; each Advisor represents a general principle that may support expert behavior [1]. This approach is supported by evidence that people integrate a variety of strategies to accomplish problem solving [2, 6, 7]. A FORR-based application is constructed for a particular set of related tasks called a *domain*, such as path finding in mazes [3] or game playing [4]. A FORR-based program develops expertise during repeated solution attempts within a *problem class*, a set of problems in its domain (e.g., contests at the same game, or trips with different starting and ending points in the same maze). FORR-based applications often produce expert-level results after learning on as few as 20 problems in a class. Learning is relatively fast because a FORR-based application begins with the pre-specified, domain-specific knowledge of its Advisors.

FORR partitions its Advisors into a hierarchy of tiers, based upon their correctness and the nature of their response. A FORR-based program begins with a set of pre-specified Advisors intended to be *problem-class independent*, that is, relevant to most classes in the domain. Each Advisor represents some domain-specific principle likely to support expert behavior. An Advisor is represented as a time-limited procedure whose input is the current problem state and the legal actions in that state. An Advisor's output is a set of *comments* which indicate how its particular principle evaluates those actions. A comment has the form <*strength, action, Advisor*> where *strength* indicates the degree of support or opposition as an integer in [0, 10]. Comments may vary in their strength, but an Advisor may not comment more than once on any action in the current state.

During execution, a FORR-based application develops expertise for a new problem class as it learns *weights* (described in Section 3.3) to reflect the reliability and utility of Advisors for that particular set of problems. Thus far, ACE learns only from its own problem solving attempts, without examples solved by others. (Our description here does not cover the full scope of the FORR architecture, only those aspects which ACE currently uses. The interested reader can find further details in [8].)

3. ACE

ACE is a FORR-based program for the CSP domain. For ACE, a problem class is the set of problems produced by the generator under a particular set of specifications, as described in Section 4. Within any given class, problems are randomly generated.

3.1 Decision Making in ACE

A problem-solving state for ACE is a partially (or fully) solved CSP. A state description pairs each variable either with an assigned value or with a domain of possible values from which its value will be selected. ACE represents the solution to a CSP as a sequence of steps generated with the following algorithm:

```
While some variable has no assigned value
    Select an unassigned variable v
    Select a value a for v from the domain of v
    Assign a to v
    For each unassigned neighbor n of v
      Remove any values in the domain of n inconsistent with a
      While any unassigned variable has an empty domain
          Retract the most recent value assignment
```

From this perspective, a CSP on *n* variables requires at least 2*n* decision steps in its solution: *n* variable selections alternated with *n* value selections. Thus an error-free solution path will contain exactly 2*n* steps.

Given this framework, there are four key decision-making processes involved: propagation, retraction, variable selection, and value selection. ACE has a set of procedures to address each process. Propagation is done either with *forward checking*, where the domains of the neighbors (in the constraint graph) of the newly-valued variable are recalculated, based on the constraints from the newly-valued variable, or with *maintained arc consistency* (MAC3), where this process is extended repeatedly to every unassigned variable until no domain changes. Retraction is currently done only with *backtracking*, which returns to a node closer to the root in the search tree, where it retracts the value assignment that caused the empty domain and removes that value from the legal values that variable may subsequently assume in the current state. If necessary, additional variables are automatically "unvalued" as well. With either propagation method plus backtracking, ACE is *complete* (capable of finding a solution). The procedures for variable selection and value selection are represented in ACE by Advisors.

3.2 ACE's Advisors

In a domain with many heuristics, FORR's Advisor hierarchy promotes both efficiency and accuracy. First, *tier 1* isolates rationales expected to be correct from those that are merely heuristic. A FORR-based application begins decision making with a pre-sequenced list of tier-1 Advisors. When a tier-1 Advisor comments positively on an action, no subsequent Advisors are *consulted* (given the opportunity to comment), and the action is executed. When a tier-1 Advisor comments negatively on an action, that action is eliminated from consideration, and no subsequent Advisor may support it. If the set of possible actions is thereby reduced to a single action, that action is executed. ACE has two tier-1 Advisors, Victory and Later. If only one variable remains unvalued and has at least one legal value, *Victory* assigns it a value. If an iteration is for variable selection, *Later* limits that choice to variables whose degree is at least as large as the number of values remaining after propagation. (Although Later is always correct for graph coloring, it is not for general CSP's. We are currently addressing this notion of timing with an approach similar to that of Section 4.3)

Typically with FORR, tier 1 does not identify an action, so control passes to *tier 2*. Tier-2 Advisors plan, and recommend sequences of actions, instead of a single action. (ACE does not yet incorporate tier-2 Advisors; they are a focus of current work.) Finally, if neither tier 1 nor tier 2 produces a decision, control passes to *tier 3*, where most decisions are made. Tier-3 Advisors are heuristic and consulted in parallel. A decision is computed as a weighted combination of their comments in a process called

Table 1. The concerns underlying ACE's tier-3 Advisors. All concerns are computed dynamically, except where noted. Sources are given where relevant.

Concern	**Definition**
Variable selection	
Degree	Number of neighbors in the constraint graph (static)
Domain	Number of remaining possible values
Forward Degree	Number of unvalued neighbors
Backward Degree	Number of valued neighbors
Domain/Degree	Ratio of domain size to degree
Constraints	Number of constraint pairs on variable [9]
Edges	Edge degree, with preference for the higher/lower degree endpoint (static)
Reverse Edges	Edge degree, with preference for the lower/higher degree endpoint (static)
Dynamic Edges	Edge degree, with preference for the higher/lower degree endpoint
Dynamic Reverse Edges	Edge degree, with preference for the lower/higher degree endpoint
Value selection	
Common Value	Number of variables already assigned this value
Options Value	Number of constraints on selected variable that include this value
Conflicts Value	Resulting domain size of neighbors [10]
Domain Value	Minimal resulting domain size among neighbors [10]
Secondary Options Value	Number of constraints from neighbors to nearly-neighbors
Secondary Value	Number of values among nearly-neighbors
Weighted Domain Size Value	Domain size of neighbors, breaking ties with frequency [10]
Point Domain Size Value	Weighted function of the domain size of the neighbors, a variant on an idea in [10]
Product Domain Value	Product of the domain sizes of the neighbors

voting, described in the next section. The action that receives the most support during voting is executed, with ties broken at random. (ACE uses heuristics to search because the problem space in which it functions is NP-hard. Nonetheless, any solution it produces is complete.)

Each of ACE's tier-3 Advisors encapsulates a single primitive, naive approach to selecting a variable or selecting a value. To generate them, we identified basic properties (*concerns*) and formulated one procedure to minimize each concern and another to maximize it. Thus each concern gives rise to two Advisors. Some concerns were gleaned from the literature, some are common CSP lore, and others were naively hypothesized. For example, one traditional concern is the *degree* of a variable, the number of neighbors it has in the constraint graph. For variable selection, the tier-3 Advisor *Max Degree* supports the selection of unvalued variables in decreasing degree order, with comment strengths from 10 down. Although Max Degree is popular among CSP solvers, we also implemented its dual, *Min Degree*, which comments on variables in increasing degree order. Another example of a concern, this time a naïve one for value selection of an already-chosen variable, is *common value*, the number of variables already assigned this value. *Min Common Value* supports the selection of values less frequently in use; *Max Common Value* is its dual. The full complement of concerns that generate ACE's tier-3 Advisors is detailed in Table 1. There, *edge degree* is the sum of the degrees of the endpoints of an edge, and a *nearly-neighbor* is a variable at distance two in the constraint graph from the variable being assigned a value.

These 19 concerns generate 38 Advisors that correspond naturally to properties of the constraint graph and search tree associated with general CSP's. A skeptical reader might be concerned that, consciously or not, we have somehow "biased" our set of Advisors. Even if that were so, we would respond that it is still up to FORR to learn how to use the Advisors appropriately, and that the ability to incorporate our expertise into the FORR architecture by specifying appropriate Advisors is a feature, not a bug.

3.3 Voting and Weight Learning in ACE

Although a FORR-based program begins with problem-class-independent, tier-3 Advisors, they may not all be of equal significance or reliability in a particular problem class. Therefore, FORR is equipped with a variety of weight-learning algorithms. FORR permits the user to partition each task into *stages*, so that a weight-learning algorithm can learn weights for each stage in the solution process. For now, the number and definition of each stage is pre-specified by the user. The premise behind all weight learning in FORR is that the past reliability of an Advisor is predictive of its future reliability. We used a weight-learning algorithm called DWL in these experiments.

DWL (*Digression-based Weight Learning*) learns problem-class-specific weights for tier-3 Advisors; it is specifically designed to encourage short solution paths. After a problem has been solved successfully, DWL examines the trace of that solution. DWL extracts *training instances*, pairs of the form <*state*, *decision*>. The intuition behind DWL is suggested by Figure 1, which diagrams the search to a solution. The solid path is the *underlying perfect search path*; it includes exactly 2*n* correct decision steps, represented in Figure 1 as black circles. Those decisions should be maximally reinforced. The remainder of the search consists of *digressions*, subtrees rooted along the solid path, whose roots (represented as white circles in Figure 1) are eventually retracted. A decision at the root of a digression is an error that produces an *over-constrained* (i.e.,

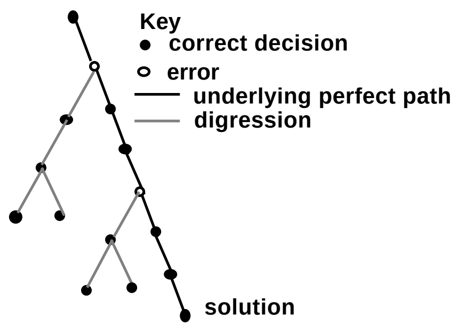

Fig. 1. A search tree for a CSP, as seen by DWL

unsolvable) sub-problem; it should be discouraged. Decisions at all but the root of a digression address an over-constrained problem, and should be reinforced in inverse proportion to the number of steps required to discover that the problem has no solution.

Under DWL, all pre-specified tier-3 Advisors begin as equally significant. The comments of an often-correct Advisor gradually have more influence during voting, while those of an often-incorrect Advisor are soon overwhelmed. DWL learns weight w_i for Advisor i with the following algorithm:

```
For all i, $w_i \leftarrow 0.05$
For each training instance <s, d>with state s, decision a, and next state s′
  For each Advisor $A_i$ that produces comments $c_i$ on s
    $d_i \leftarrow d_i + 1$
    If s′ was not the root of a digression
       then if $c_i$ supports a, increase $w_i$ else decrease $w_i$
       else if $c_i$ supports a, decrease $w_i$ else increase $w_i$
```

DWL also uses a *discount factor* of 0.1 to introduce each Advisor gradually into the decision process. When ACE makes a decision, tier 3 chooses the action with the greatest support, as follows:

$$\underset{j}{argmax} \left\{ \sum_i \omega_i w_i c_{ij} \right\} \text{ where } \left\{ \begin{array}{l} d_i = \text{number of opinions } i \text{ has generated.} \\ \omega_i = \begin{cases} 0.1 * d_i \text{ if } d_i < 10. \\ 1 \text{ otherwise.} \end{cases} \\ w_i = \text{weight of Advisor } i. \\ c_{ij} = \text{opinion of consulted Advisor } i \text{ on choice } j. \end{array} \right.$$

Note that an Advisor must be consulted and comment to figure in this computation. Although all Advisors are consulted during learning, only those that have earned a weight greater than that of *Anything* are consulted during testing. Anything is a non-voting baseline Advisor which comments with randomly-generated strength on *n* randomly-chosen actions $(0.5)^n$% of the time. Thus DWL fits ACE to correct decisions, learning to what extent each tier-3 Advisor reflects expertise.

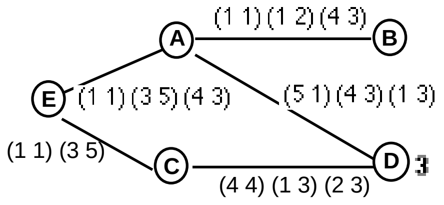

Fig.2. A partially-valued problem on 5 variables. Edges are labeled by permissible value pairs, in alphabetical order

3.4 An Example

The following example shows how ACE makes decisions. Figure 2 represents a constraint graph on five variables, each with domain {1, 2, 3, 4, 5}. If D were now assigned the value 3, D's immediate neighbors (A and C) have their own domains reduced: A can now be 1 or 4, while C can be 1 or 2. Under forward checking, ACE does not continue to remove values of variables more distant from D, so B and E would still have 5 possible values each, while A and C would have 2. ACE must now select another variable to value. Since the number of possible values for each of B and E is greater than their respective degrees, the Advisor Later will eliminate B and E from among the possibilities. The remaining unvalued variables, A and C, are then forwarded to the variable-selection Advisors in tier 3. For example, Min Degree would support the selection of variable C with a strength of 10, and the selection of variable A with a strength of 9. (In this simple example, Max Degree would counter those comments exactly. In a larger constraint graph, however, the dual pairs of Advisors typically address different choices.) Similarly, Max Forward Degree would support the selection of variable A, which has 2 unvalued neighbors, with a strength of 10, and variable C, which has one unvalued neighbor, with a strength of 9. ACE tallies the comments on A and C from all the tier-3 Advisors, multiplying each comment's strength by the weight of the Advisor that produced the comment, and then selects the variable with the most support.

4. Experimental Design and Results

Because it breaks voting ties at random, ACE is non-deterministic. Its performance is therefore typically judged over a set of *r* runs. Each *run* consists of a learning phase and a testing phase. In the *learning phase*, the program learns weights while it attempts to solve each of *l* problems from the specified class. In the *testing phase*, weight-learning is turned off, and the program tries to solve a set of additional problems from the same class. Since ACE can get stuck in a "blind alley," where there are no successes from which to learn, new runs also present fresh opportunities for success. This is actually conservative, as we argue below that one could reasonably utilize the best result from multiple runs.

In all the experiments reported here, ACE used DWL. Rather than assume that selection heuristics are consistently reliable throughout the solution of a problem, we specified three distinct weight-learning stages, determined by the percentage of vari-

Table 2. Performance of an ablated version of ACE under forward checking, averaged over 10 runs in different training environments. All problems had 30 variables, maximum domain size 5, tightness .4, and density .005. Power is percentage of backtrack-free solutions. Time is in seconds per solved problem. Also included are the number of steps in the longest solution during testing, and the percentage of learning problems unsolved due to the step limit.

Learning step limit	Power	Time	Checks	Longest testing solution	Unsolved learning problems
70	13%	.16	145.40	1639	32.38%
80	14%	.12	118.44	717	21.38%
100	**18%**	**.13**	**167.77**	**1041**	**11.50%**
200	12%	.17	163.34	2471	5.00%
400	10%	.13	125.41	837	1.25%
800	11%	.12	121.93	511	0.50%
1000	11%	.14	125.68	1035	0.38%

ables thus far assigned values: *early* (fewer than 20%), *middle* (at least 20% but no more than 80%), and *late* (more than 80%). (This is different from the graph coloring work in [5] which employed only a single stage.) Unless otherwise stated, ACE also used MAC3 for propagation and was permitted no more than *s* task steps per problem. A *task step* is either the selection of a variable, the selection of a value, or the retraction of a value or a variable. As discussed earlier, a CSP on *n* variables requires at least 2*n* steps. Data was averaged over 10 runs, each with a learning phase of 80 problems followed by a testing phase of 10 problems.

All problems were produced by a random problem generator available at http://www.cs.unh.edu/ccc/code.html. Although there is no guarantee that any particular set of problems was distinct, given the large size of the problem classes the probability that a testing problem was also a training problem is extremely small. Our generator defines *edge density* as the percentage of possible edges beyond the minimal *n*-1 necessary to connect the constraint graph on *n* vertices, and *tightness* as the percentage of possible value pairs that are forbidden by the constraints. When generating problems of a fixed size (number of variables, maximum domain size, and tightness), increasing the density eventually makes it more difficult to find problems with solutions. For example, with 50 variables, domain size 20, and tightness 0.6, no such problems could be generated with density 0.100 in 100 attempts, although they could be generated at density 0.080. We focus our attention here on densities where problems with at least one solution were readily generated. (Those reported upon here are guaranteed to have at least one solution, but ACE can work with over-constrained problems as well.)

We evaluated ACE on its solutions: by computation time (in seconds), number of retractions (backtracks), and number of constraint checks. (A *constraint check* is a confirmation that a value in the domain of an unassigned variable is still possible given the current assigned values.) Cited differences are significant at the 95% confidence level.

4.1 Step-limited Learning

The learning phase in a run gleans training instances from successful problem-solving experience, as described in Section 3.3. ACE's learning is, at the moment, *unsupervised*, that is, no outside expert is offering suggestions or corrections. The quality of its training instances is unpredictable — they may be good examples or they may simply have been lucky choices. To function as an expert, however, ACE needs to experience the sorts of decision situations that an expert would confront.

One way to enhance the quality of the training instances is to consider only those from good (i.e., short) solution traces. ACE can either abandon a problem after some fixed number of steps, or work until it is solved. The experiment described here sought to identify the best approach. We used forward checking with a primitive version of ACE that included only the first four concerns from Table 1, so that there would be room for improvement in the program's performance. We then limited the program to *s* = 70, 80, 100, 200, 400, 800, and 1000 steps for learning, but permitted it to work to solution during testing. The results appear in Table 2, where *power* is percentage of backtrack-free solutions. Although their size (30 variables, domain size 5) leaves room for inefficient solutions by naïve solvers, these are sparse problems (density .005, tightness .4), so one would expect an expert to solve them quickly. A perfect solution to such a problem would include 60 steps. Under a 70-step limit, however, ACE did not solve, and therefore could not learn from, nearly a third of the learning problems. Moreover, its performance after learning on the remaining two thirds offered no noteworthy improvement in time or number of constraint checks over higher *s* values. When the program was given many more than 100 steps in which to solve learning problems, the quality of its testing performance never consistently improved. We have therefore settled on a 100-step learning limit for problems on 30 variables; it is efficient during development and appears to provide examples as good as those from higher or lower limits. During testing, however, we let each program run until it solved the problem.

4.2 The Value of Learning

The next set of problems, this time for the full version of ACE, all had a larger domain size of 8 and tightness .5. Densities of 0.075 and 0.100 for 30 variables and .045 for 50 variables were examined. For comparison, on 100 problems from each problem class,

Table 3. ACE's performance, as averaged over 10 runs. All problems had maximum domain size 8, and tightness .5. Power is percentage of backtrack-free solutions.

Variables	Density	Approach	Power	Time	Retractions	Checks
30	.075	Traditional	68%	0.43	0.41	7381.90
30	.075	ACE no learning	87%	0.99	0.19	6323.31
30	.075	ACE learning	70%	0.72	0.60	6884.24
30	.100	Traditional	19%	0.65	3.42	12216.55
30	.100	ACE no learning	61%	1.72	6.40	12945.50
30	.100	ACE learning	81%	0.96	1.24	8476.14
50	.045	Traditional	0%	0.16	3.50	39445.70
50	.045	ACE learning	42%	2.00	3.89	21599.17

we also tested both ACE without weight-learning and *Traditional*, which minimizes the ratio of a variable's domain size to its degree and selects values at random [5]. (Traditional simply reduces ACE to a single tier-1 Advisor.)

The results, in Table 3, show that weight learning substantially improved performance on the two more difficult sets of problems. With learning, ACE solved those test problems with fewer constraint checks, and in less time. ACE requires more elapsed time than Traditional because its decision calculation is more complex: it must compute the opinions of many Advisors, combine them with their current weights, and tally the results. The resultant decisions, however, are more likely to be correct, that is, not

Table 4. The number of runs during which an ACE tier-3 Advisor was active during early (1), middle (2), and late (3) stage testing on the last two problem sets in Table 3. Particularly consistent Advisors appear in bold, those that are significant without their dual appear in italics, and those that are infrequently relevant appear in parentheses.

	n = 30			*n* = 50		
Advisor	1	2	3	1	2	3
Variable concerns						
Max Backward Degree		**10**	**10**		**10**	**10**
(Min Backward Degree)				2		
Max Constraints	**10**		2	**10**		
Max Degree	**10**	**10**	**10**	**10**	**10**	**10**
Min Domain	7	**10**	**10**	**10**	**10**	**10**
Min Domain/Degree	**10**	**10**	**10**	**10**	**10**	**10**
Max Dynamic Edges	**10**		7	**10**		
Max Dynamic Reverse Edges			7			**10**
Max Forward Degree	**10**		4	**10**		
Min Forward Degree			6			**10**
Max Reverse Edges	**10**	5	9	**10**		
Value concerns						
Max Common Value	3	7	4	1	8	8
Min Common Value	**10**	6	6	9	7	2
Max Conflicts Value		2	5		4	4
Min Conflicts Value	**10**	8	5	9	6	6
Max Domain Value	**10**	8	5	**10**	6	6
Min Domain Value		2	5		4	4
Max Options Value	**10**	8	5	9	6	6
Min Options Value		2	5		4	4
Max Point Domain Size Value		2	5		4	4
Min Point Domain Size Value	**10**	8	5	**10**	6	6
Max Product Domain Value	**10**	8	5	**10**	6	6
Min Product Domain Value		2	5		4	4
Max Secondary Options Value	**10**	8	5	**10**	6	6
Min Secondary Options Value		2	5		4	4
Max Secondary Value	**10**	8	5	**10**	6	6
Min Secondary Value		2	5		4	4
Max Weighted Domain Size Value	**10**	8	5	**10**	7	6
Min Weighted Domain Size Value	**10**	2	5	**10**	5	4

retracted later.

Recall that an Advisor active during testing is one whose weight has demonstrated that it consistently performs better than random advice (Anything). For each class, the Advisors that ACE relied on during testing appear in Table 4, along with the number of runs (out of 10) where their weight in a stage qualified them for use during testing. Together with their weights, the Advisors of Table 4 could be construed as an algorithm for variable selection and value selection within the problem class. Note that al-though every concern is represented in Table 4, 9 Advisors are absent. According to Table 4, both an Advisor and its dual (e.g., Max and Min Weighted Domain Size Value) can be relevant in a stage; it is the extremity of the concern, rather than its value, that is significant. Some Advisors, such as Max Constraints, are relevant only in a single stage. Others, such as Max Dynamic Edges, are relevant only early and late in problem solving, and not in the middle. Still others, such as Max Degree, appear consistently relevant. The active Advisors also display much greater consistency in the early stage of problem solving. Finally, there appear to be more heuristics that are consistently reliable in all stages for variable selection than there are for value selection.

It is interesting to compare ACE's learned weights with the performance predicted in the literature for some of these concerns. Min Domain/Degree [11] is confirmed as valuable in all problem-solving stages. Min Point Domain Size Value is confirmed for early and middle-stage solving, but relevant only about half the time for late-stage solving. Contrary to prediction, however, the Constraints concern functions well only in the early period, and as a maximum, not as a minimum. This may turn out to be a peculiarity of this class of problems, but one of ACE's advantages is that it automates the process of adapting heuristic processing to specific problem classes.

4.3 Learning New Advisors

Given a language in which to express them, FORR can learn new tier-3 Advisors [12]. Min Domain/Degree led us to wonder about the efficacy of other arithmetic combinations of concerns. We therefore formulated an *arithmetic variable language* in which an Advisor's concern would be either the product or the quotient of a pair of variable-selection concerns. (Sums and differences proved weaker in early testing and were eliminated.) Each expression in this language is of the form:

$$<\text{function}_1, \text{function}_2, \text{attitude}, \text{stage}>$$

where stage is early, middle, or late (as defined in Section 3.3), the two functions are any pair of distinct concerns (such as those in Table 1), and *attitude* is one of the following: maximize $\text{function}_1 * \text{function}_2$, minimize $\text{function}_1 * \text{function}_2$, maximize $\text{function}_1 / \text{function}_2$, and minimize $\text{function}_1 / \text{function}_2$.

To learn new Advisors within a language, FORR monitors how each possible expression would perform at problem solving. During weight learning, each expression is given the opportunity to comment on training instances from the underlying perfect search path, as if it were a tier-3 Advisor. For each expression, FORR tallies each attitude's frequency (number of times it discriminates) and accuracy (number of times it is correct) for each stage.

During problem solving, each expression in a language is either potential, active, spawned, or inactive. Initially, all expressions are *potential*, monitored for possible inclusion in decision making. After every t tasks (t = 10 here), FORR reevaluates each

Table 5. Performance of a variety of programs, with and without the ability to learn tier-3 Advisors. MDD is the program with Min Domain/Degree as its only Advisor. ACE– is the program without the Domain/Degree concern. The learning Advisors approach is an ablated version of ACE, as described in the text.

Approach	Concerns	Power	Time	Retractions	Checks
MDD	—	19%	0.69	3.42	12216.55
ACE	all	81%	0.96	1.24	8476.14
ACE–	all but MDD	80%	0.96	1.17	8152.18
Learn Advisors	4 variable	80%	1.04	1.34	8048.66

expression's status. Those that fail to discriminate or never comment are eventually made *inactive*, to speed subsequent computation. Potential expressions with an attitude that has been accurate at least 85% of the time are promoted to *active* status. Active expressions provide input to their language's *summary Advisor*, which combines their comments to structure its own. Once an active expression has tallied 95% accurate, it is *spawned*, that is, transformed into an individual Advisor, and the expression no longer participates in the summary Advisor's computation. Both the summary Advisor and the spawned Advisors are subject to the discount factor discussed in Section 3.3, so that they enter the decision process gradually. This permits ACE to maintain its performance level as it introduces new Advisors.

In our first experiment with learning new Advisors, we formulated a simple language for the concerns Domain, Degree, Forward Degree, and Backward Degree. As described earlier, each expression combined a pair of these concerns (e.g., Domain and Degree) and considered four computations on them: minimize their product, maximize their product, minimize their quotient, or maximize their quotient. We then removed the Domain/Degree concern from ACE's list in Table 1 (*ACE–*), and had ACE learn on 30 variables, maximum domain size 8, tightness .4, density .100. For comparison we also tested ACE– alone, and *MDD*, a program with Min Domain/Degree as its only Advisor.

The results, in Table 5, were startling. ACE outperformed MDD, as expected, but so did ACE–, suggesting that the Min Domain/Degree heuristic might not make a necessary contribution, despite its high weight. Indeed there was no statistically significant difference between ACE and ACE– along any metric. Furthermore, when ACE was permitted to learn new Advisors in a language capable of expressing Min Domain/Degree (the last line in Table 5), the expression for Min Domain/Degree never became active even once in 10 runs. Instead, on every run, exactly one expression ever became active, the same one every time: "maximize the product of degree and forward degree in the early stage." With this single learned Advisor, the learning-Advisors version of ACE performed just as well as ACE and as ACE–.

4.4 Learning Transfer

We have confirmed, outside of ACE and on fairly hard problems, that when the new "maximize the product of degree and forward degree" heuristic is used at the top of the search tree, subsequent use of Min Domain/Degree is comparable to the use of domain size alone. To accomplish this, we incorporated the new heuristic into a CSP algorithm

Table 6. Performance results in nodes per problem for MAC3 and three conventional heuristics, with and without the product heuristic learned by ACE. Data is averaged over 10 runs using code separate from ACE. Problems had 150 variables, domain size 5, density .05, and tightness .24.

Conventional heuristic	Alone	Enhanced heuristic
Min domain (MD)	86,065	3,218
MDD	4,277	3,218
MD after degree preorder	12,602	3,218

coded in a conventional manner, and tested it on reasonably difficult 150-variable problems.

Our results, in Table 6, are significant for several reasons. They demonstrate that:

- Lessons learned with ACE can be transferred to a conventional algorithmic context.
- Lessons learned on easy problems can be relevant to hard problems.
- Our conventional understanding of search-order heuristics may be overly simplistic.

Our initial tests ran three conventional heuristics: min domain (MD), min (domain/degree) (MDD), and MD after first ordering the variables by descending degree. Then we repeated the same tests, this time replacing the conventional heuristic by min (degree * forward-degree) in the top fifth of the search tree. It is admittedly odd that all the latter tests averaged to the same (rounded off) search tree size, but the top of the tree appears so dominant that, in most cases, the same nodes get visited, albeit in a different order. With this approach, the search tree is actually somewhat reduced, while processing time slightly increases due to the dynamic calculation of forward degree.

In general, the importance of the processing at the top of the search tree is not surprising, but ACE allows us to make progress on turning "folklore" (what you do at the top of the search tree is more important than what you do at the bottom) into science (or, at least, into engineering). The fact that domain size, the conventional bedrock of variable ordering, can be ignored at the critical top of the search tree is surprising, at least at first blush. On reflection it would seem to make perfect sense that domain size would be less critical at the top of the tree, before propagation from search choices has as much chance to effect domain size reduction, while forward degree would be critical at the top of the tree, where it is going to be relatively large, and help to determine the amount of propagation. ACE can inspire us to pose and help us to further evaluate such hypotheses.

5. Discussion

A traditional CSP heuristic for variable ordering uses secondary heuristics to break ties, or combines a few heuristics in crude mathematical combinations. The few attempts to automate the construction of constraint solvers, including Laurière's pioneering work, have thus far either tried to select a single method or tried to invent a special-purpose algorithm [13-17]. In contrast, ACE permits us to order such advice in a more flexible and subtle manner.

In experiments on a variety of problems, ACE regularly identifies heuristics previously considered essential by constraint researchers in a general CSP context: Min Domain, Max Degree, Min Domain/Degree, and Max Backward Degree. In particular, for ordinary CSP’s, ACE has confirmed the importance of minimal domain size, often employed individually by constraint researchers in a general CSP context, as a variable ordering heuristic. ACE indicates, however, that in the early stage, when fewer than 20% of the variables have been valued, Min Domain is not necessarily productive.

With Advisor learning, ACE permits thorough empirical investigation not only of individual heuristics, but of combinations of them as well. ACE’s ability both to confirm conventional wisdom and to provide further analysis is enticing. The emphasis on maximizing backward degree, for example, is puzzling until one realizes that, since these problems are so sparse, maximizing backward degree may push a significant number of degree-one variables to the end of the search, where forward checking will have ensured that they can be instantiated without any backtracking. This raises a further puzzle as to why minimizing backward degree does not figure prominently in the late stage; but then one realizes that this could risk switching over too early. Maximizing backward degree will do the “right “thing up to and including the time when the maximum backward degree is one, whenever that occurs. All this begins to seem awfully clever of ACE, though, of course, “clever” is not really the operative word here; the currently popular “emergent” may be more appropriate.

In general, ACE is able not only to “rediscover” useful heuristics, but also to explore more sophisticated combinations and timing patterns in applying these heuristics than an individual experimenter could easily consider. In this manner ACE can not only support but also instigate research. Our work with Domain/Degree in Section 4.3 is a good example.

To build ACE we did not have to tune FORR’s learning parameters. The discount factor of 0.1 has been traditional in FORR, as have the initial Advisor weight of 0.05 and the constants employed in learning new Advisors (discussed in Section 4.3). The proportion of Anything’s comments was devised to make few comments more likely than a single one, and again is standard in FORR. Only the stage designations (set rather arbitrarily at 20% and 80%) are new; in previous FORR-based, non-CSP applications there were only two stages, with 15% in the early stage.

Our current research plans include extension to over-constrained problems, as well as to other, more concrete classes of CSP’s. We are also investigating a variety of propagation and retraction methods, and actively solicit empirically-validated suggestions for new Advisors from the CSP community. Furthermore, we are working to make stage designation more flexible, and to incorporate other weight learning algorithms, as well as planning Advisors for tier 2.

In summary, ACE is intended to become a comprehensive architecture for acquiring and controlling collaborative and adaptive constraint solving methods. It will establish a taxonomy of Advisors, from very problem-dependent to very general, languages in which to express them, and ways to combine them effectively. ACE should eventually be able to acquire uncodified expertise, to uncover new techniques, and to discover useful new solvers for specific classes of CSP’s. In short, ACE is both a CSP-solver and a partner in CSP research.

Acknowledgements: We thank Francesca Rossi and Alessandro Sperduti for suggestions that led to DWL. We also thank the anonymous referees for their constructive suggestions. Eugene Freuder is supported by a Principal Investigator Award from Science Foundation Ireland.

References

1. Epstein, S. L.: For the Right Reasons: The FORR Architecture for Learning in a Skill Domain. Cognitive Science 18 (1994) 479-511
2. Ratterman, M. J. and Epstein, S. L.: Skilled like a Person: A Comparison of Human and Computer Game Playing. In Proceedings of Seventeenth Annual Conference of the Cognitive Science Society. Lawrence Erlbaum Associates, Pittsburgh (1995) 709-714
3. Epstein, S. L.: On Heuristic Reasoning, Reactivity, and Search. In Proceedings of Fourteenth International Joint Conference on Artificial Intelligence. Morgan Kaufmann, Montreal (1995) 454-461
4. Epstein, S. L.: Prior Knowledge Strengthens Learning to Control Search in Weak Theory Domains. International Journal of Intelligent Systems 7 (1992) 547-586
5. Epstein, S. L. and Freuder, G. Collaborative Learning for Constraint Solving. In: Walsh, T. (ed.): Principles and Practice of Constraint Programming -- CP2001. Vol. 2239. Springer Verlag, Seattle WA (2001) 46-60
6. Biswas, G., Goldman, S., Fisher, D., Bhuva, B. and Glewwe, G. Assessing Design Activity in Complex CMOS Circuit Design. In: Nichols, P., Chipman, S. and Brennan, R. (eds.): Cognitively Diagnostic Assessment. Vol. Lawrence Erlbaum, Hillsdale, NJ (1995)
7. Crowley, K. and Siegler, R. S.: Flexible Strategy Use in Young Children's Tic-Tac-Toe. Cognitive Science 17 (1993) 531-561
8. Epstein, S. L.: Pragmatic Navigation: Reactivity, Heuristics, and Search. Artificial Intelligence 100 (1998) 275-322
9. Kiziltan, Z., Flener, P. and Hnich, B.: Towards Inferring Labelling Heuristics for CSP Application Domains. In Proceedings of KI'01. Springer-Verlag, (2001)
10. Frost and Dechter, R.: Look-ahead value ordering for constraint satisfaction problems. In Proceedings of IJCAI-95. Montreal (1995) 572-578
11. Bessiere, C. and Regin, J.-C. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In: Freuder, E. C. (ed.): Principles and Practice of Constraint Programming - CP96, LNCS 1118. Vol. Springer-Verlag, (1996) 61-75
12. Epstein, S. L., Gelfand, J. and Lock, E. T.: Learning Game-Specific Spatially-Oriented Heuristics. Constraints 3 (1998) 239-253
13. Borrett, J., Tsang, E. P. K. and Walsh, N. R.: Adaptive constraint satisfaction: the quickest first principle. In Proceedings of 12th European Conference on AI. Budapest, Hungary (1996) 160-164
14. Caseau, Y., Laburthe, F. and Silverstein, G. A Meta-Heuristic Factory for Vehicle Routing Problems. In: Principles and Practice of Constraint Programming – CP’99. Vol. LNCS 1713. Springer, Berlin (1999)
15. Minton, S.: Automatically Configuring Constraint Satisfaction Programs: A Case Study. Constraints 1 (1996) 7-43
16. Smith, D. R. KIDS: A Knowledge-based Software Development System. In: Lowry, M. R. and McCartney, R. D. (eds.): Automating Software Design. Vol. AAAI Press, (1991)
17. Laurière, J. L.: ALICE: A Language and a Program for Solving Combinatorial Problems. Artificial Intelligence 10 (1978) 29-127