Computational Intelligence, Volume 21, Number 4, 2005

LEARNING TO SUPPORT CONSTRAINT PROGRAMMERS

SUSAN L. EPSTEIN

Department of Computer Science, Hunter College and The Graduate Center of The City University of New York, New York, USA

EUGENE C. FREUDER AND RICHARD J. WALLACE

Cork Constraint Computation Centre, University College Cork, Cork, Ireland

This paper describes the Adaptive Constraint Engine (ACE), an ambitious ongoing research project to support constraint programmers, both human and machine. The program begins with substantial knowledge about constraint satisfaction. The program harnesses a cognitively oriented architecture—FOr the Right Reasons (FORR) to manage search heuristics and to learn new ones. ACE can transfer what it learns on simple problems to solve more difficult ones, and can readily export its knowledge to ordinary constraint solvers. It currently serves both as a learner and as a test bed for the constraint community.

Key words: constraint satisfaction, machine learning, mixture of experts, cognitively oriented architecture.

1. INTRODUCTION

Many large-scale, real-world problems are readily represented, solved, and understood as constraint satisfaction problems (CSPs). Constraint programming offers a wealth of good, general-purpose methods to solve problems in such fields as telecommunications, Internet commerce, electronics, bioinformatics, transportation, network management, supply chain management, and finance (Nudel 1983; Freuder and Mackworth 1992). As a result, organizations throughout the world already exploit CSP technology to solve difficult problems in design and configuration, planning and scheduling, and diagnosis and testing. Yet each new, large-scale CSP faces the same bottleneck: difficult constraint programming problems need people to "tune" a solver efficiently. Armed with hard-to-extract domain expertise, scarce human CSP experts must now select, combine, and refine the various techniques currently available for constraint satisfaction and optimization.

CSP solution remains more art form than automated process, in part because the *interactions* among existing CSP methods are not well understood. There is increasing evidence to suggest that different classes of CSPs respond best to different heuristics (Borrett, Tsang, and Walsh 1996), but arriving at appropriate methods in practice is not a trivial cookbook exercise (Beck, Prosser, and Selensky 2003). At present, for each new, large-scale CSP, a constraint programmer must seek an effective program based upon the right method combination.

The thesis of our work is that a program can learn to synthesize, from generic components, effective programs adapted to specific CSP problem classes. This paper reports on initial results with the Adaptive Constraint Engine (ACE) . We do not propose ACE as a substitute for any particular constraint-solving program, but as a colleague in research. ACE can support constraint programmers in their quest for method combination appropriate to a particular class of problems specified by the user. (Throughout this paper, we distinguish carefully between the programmer, who writes code, and the user, who merely submits experiments to ACE for execution using that code.) ACE can support a novice constraint programmer in the selection of heuristics. It can learn new, efficient heuristics, those that were previously unidentified by experts and can be readily used by them in other programming environments. ACE can learn heuristics for problem classes that do not succumb to the ordinary, off-the-shelf CSP approaches. Thus we do not pit the program against others, but show results that improve problem solving, provide insight, and/or export to other solvers.

© 2005 Blackwell Publishing, 350 Main Street, Malden, MA 02148, USA, and 9600 Garsington Road, Oxford OX4 2DQ, UK.

The principal results of this paper with respect to ACE are that:

- ACE learns to solve difficult CSPs efficiently.
- ACE characterizes different classes of CSPs differently.
- ACE learns heuristic combinations for graph coloring that correspond to well-known methods.
- ACE has learned new heuristics that readily export to improve ordinary CSP solvers.

Moreover, the work reported here demonstrates that it is possible for a weighted mixture of expert systems to adapt itself to a problem environment in a number of ways:

- To learn weights for its experts
- To learn how to prioritize its experts
- To learn new experts
- To learn when to ignore its experts entirely
- To learn when to stop learning

As a result, ACE's learning can provide guidance in problem classes where ordinary CSP approaches stumble. Noteworthy for those whose interests lie in reasoning mechanisms, ACE can determine when it has finished learning, and can learn when and how to modify its own reasoning structure to improve its performance.

ACE addresses expertise not over a particular set of problems, but over an entire problem-solving paradigm. ACE is built within a general learning and problem-solving architecture called FORR, discussed in Section 4. FORR (FOr the Right Reasons) relies on substantial knowledge, both declarative and procedural. The programs previously built within FORR were intended to learn to solve related problem classes, such as games (Epstein 2001) or mazes (Epstein 1995).

The next two sections describe the inherent difficulties in learning for constraint solving, and sketch some traditional CSP solution techniques. Subsequent sections describe ACE's reasoning mechanism and how it learns, report on ACE's successes, and sketch its knowledge representation. The paper concludes with a discussion and plans for future work.

2. THE CHALLENGE OF CONSTRAINT PROBLEMS

A CSP consists of a set of variables, each with a *domain* of values, and a set of *constraints* that specify which combinations of values are allowed (Tsang 1993). (For simplicity, we restrict discussion to binary CSPs, where each constraint involves no more than two variables. Hereafter, "CSP" should be read as "binary CSP.") A *solution* for a CSP is a set of value assignments, one for each variable, that satisfies all the constraints. Every CSP has an underlying *constraint graph*, which represents each variable by a vertex. An edge in the constraint graph appears between two vertices whenever there is a constraint 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. A simple example of a CSP and its underlying constraint graph appears in Figure 1.

Many specializations of CSPs form particularly interesting problem classes:

- *Graph coloring* is a kind of CSP where all the domains are initially the same set of colors, and the constraint on every edge is "not equal."

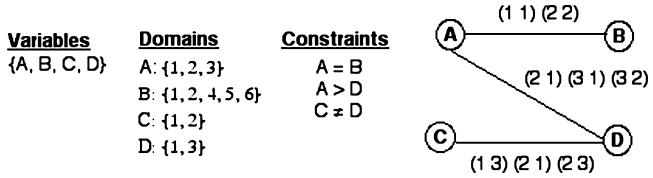


FIGURE 1. A simple constraint problem and its underlying constraint graph. Labels on the edges give acceptable values for the variables in alphabetical order. For example, (3 1) on AD means A can be 3 when D is 1. This problem has exactly one solution.

- A *geometric CSP* is formed from a random set of points in the Cartesian plane—each point becomes a variable in the problem; constraints are formed among any pair of variables within a specified (parameterized) distance of each other, with additional constraints added to connect the underlying constraint graph (Johnson et al. 1989). The result is a constraint graph ridden with clusters of vertices (not necessarily cliques), which proves particularly difficult for traditional solvers.
- Many real-world problems, such as register allocation and time-tabling, have a *small-world* topology that is different from the average randomly generated graph with the same number of vertices and edges. The *characteristic path length* of a graph is the length of the shortest path between two vertices, averaged over all pairs of vertices. The *clustering coefficient* of a graph is the average fraction of edges allowed among the neighbors of each vertex. In a small world graph, the ratio of clustering coefficient to characteristic path length is much higher than average. Intuitively, a high-proximity ratio makes the impact of each value selection greater (Walsh 1999).
- *Quasigroup* problems are Latin square problems, that is, in an $n \times n$ quasigroup each of n^2 variables participates in two cliques of size n. Quasigroup problems *with holes* eliminate some (parameterized) subset of those constraints by specifying values for some of the variables (the unspecified values are called "holes").

Geometric problems and small-world problems model data that appears in some real-world problems, where the constraint graph has many subgraphs that are cliques, or nearly cliques. Quasigroups with and without holes model certain scheduling problems.

Four parameters generally characterize a CSP: $<n, k, d, t>$. Here, n is the number of variables in the CSP, and k the maximum domain size. The *density* d of a CSP is the fraction of possible edges it includes. For a CSP with e edges, that is $2e/n(n-1)$. The *tightness* t of a CSP is the percentage of possible value pairs it *ex*cludes. For a CSP with maximum domain size k, the maximum number of admissible value pairs in a constraint is k^2. The set of all problems with the same values for n, k, d, and t forms a *class* of CSPs. Thus the CSP in Figure 1 is in the class $<4, 5, 0.5, 0.32>$. ACE has available to it a large library of such parameterized problem classes, each represented by thousands of randomly generated examples. The construction of such a library requires considerable computing resources, because problems are solved as they are generated.

A variety of programs that generate random problems within a specific class are readily available. Such generators can produce problems with one, several, or no solutions. A problem with low density and tightness is likely to be *underconstrained* and typically admits multiple solutions, while one with high density and tightness is likely to be *overconstrained* and is likely to have no solutions. Although solving a CSP is NP-hard, the most difficult problems for a fixed number of variables and domain size are those that generally lie within a relatively

narrow range of pairs of values of density and tightness (Cheeseman, Kanefsky, and Taylor 1991), known as the *phase transition*. The best-known estimate of difficulty for a CSP class is κ (Gent et al. 1996). κ estimates the constrainedness of a set of problems as the probability that a random assignment of values to some of their variables will be extendible into a solution. Problem classes with κ near one are said to be at the *complexity peak*, that is, they generally have a single solution that is particularly difficult to find.

Machine learning is, of course, predicated on the premise that the training set is sampled in a way that reflects the problem space. Ideally, problem difficulty is normally distributed across a problem space, with tails that decrease exponentially. In some important spaces, however, such as propositional satisfiability and constraint satisfaction problems, that is not the case (Gomes et al. 2000). Instead, problem difficulty has a *heavy-tailed distribution* whose tails decay according to a power law (i.e., have a Pareto-Lévy form), rather than exponentially. As a result, the constraint community often measures performance by medians rather than means, to avoid the influence of the distribution tails. Because κ is a function only on n, k, d, and t, and every problem in a class shares the same values, they have the same value for κ. At present, there is no known way to distinguish the more difficult problems from the less difficult ones within the same class, other than to solve them (Ruan, Horvitz, and Kautz 2004). As a result, a randomly selected training set may consist of problems that do not adequately, or proportionately, reflect the class. Moreover, no matter how large the training set and how well a learning program solves problems in it, the difficulty distribution guarantees that the program will eventually confront, during both learning and testing, several problems far more difficult than those it previously encountered.

Not only is the difficulty of a particular problem within a class unpredictable prior to solution, but also it is not possible to detect, without search, how many (if any) solutions a problem has. Some difficult problems admit multiple solutions, and some easy ones have none at all. Furthermore, if one succeeds in finding a solution to a particular problem, there is no way to ensure that the search was the most efficient possible path to it, nor that there was not some different, easier-to-reach solution. Thus, learning within a problem class, particularly without exploring the entire search space for each problem, is of necessity nonuniform and noise-ridden.

3. THE LEARNING TASK

A classic paradigm for CSP solution is the traditional state space search algorithm in Figure 2: repeatedly select a variable and assign it a value consistent with its constraints, until every variable has a consistent assignment. If the problem is solvable and only a single solution is required, this process requires at least n assignments. If the problem is overconstrained, or if all solutions are sought, this process requires full exploration of the search space.

```
Until the problem is solved
   Select an unvalued variable v
   Assign v the value a
   Adjust the domains of all unvalued variables *inference method*
   If domains of all unvalued variables remain non-empty
      then continue
      else return to a previous alternative value *retraction method*
```

FIGURE 2. An algorithm to find a single solution for a CSP.

During such search, a CSP solver can apply a variety of inference and retraction methods. When a *partial assignment* (a set of values assigned to a proper subset of the variables) is incompatible with the constraints, the entire subtree rooted at the partial assignment may be pruned from the search. An *inference* method propagates the implications of a newly assigned value onto the remainder of the as-yet-unassigned variables. Different amounts of inference are possible, and there are tradeoffs between inference effort and search savings. A specific inference method (the central one is arc consistency) can be carried out to varying degrees (Sabin and Freuder 1997) and with different methods (Bessière and Régin 2001). A *retraction* method is a response to an inconsistent partial assignment. Thus Figure 2 can be paraphrased as "Search the space of partial assignments of the problem, pruning forward with an inference method and retreating with a retraction method."

For a solvable CSP, the order in which one selects variables (*variable ordering*) can speed solution, as can the order in which one assigns a value (*value ordering*) to a just-selected variable. In the problem of Figure 1, for example, a good variable ordering is (A D B C), and a good value ordering for A is (2 1). There are dozens of variable-ordering and value-ordering heuristics in the CSP literature, but their interactions are ill-understood. Each heuristic relies on knowledge about what to compute, but there is little guidance as to which problem classes it works well on, whether or not it is valid throughout a single solution (or only at certain depths in the search tree), whether or not its opposite might also be valid, or how heuristics can be combined.

ACE's task is to learn to solve a specified class of problems well. The standard CSP performance criteria are applied throughout this paper: time, retractions, nodes, and constraint checks. *Time* is CPU seconds devoted to the solution of a single task. (Any learning occurs after a task and is not included in this measure.) A *retraction* is the withdrawal of a value assignment because it has proved inconsistent with the constraints and the partial assignment. A *node* is a partial or full assignment that is constructed during search. The search space of full and partial assignments for a problem in $<n, k, d, t>$ is $O(k^n)$. Finally, a *constraint check* confirms that a pair of values is acceptable between two, mutually constrained variables. Constraint checks are the classic CSP measure of work.

ACE begins with constraint solving knowledge (described in Section 7), including chronological backtracking as a retraction method, two constraint inference methods (forward checking and arc consistency), and a set of CSP ordering heuristics. What ACE learns is additional heuristics, and how to organize all its heuristics to solve a particular class of problems well.

4. THE REASONING MECHANISM: FORR

ACE is based on FORR (FOr the Right Reasons), a cognitively oriented architecture for learning and problem solving. (This cognitive orientation is discussed in Section 8.2.) FORR is a *mixture of expert* decision makers, a system that combines the opinions of a set of expert-like procedures to make a decision (Chatterjee and Chatterjee 1987; Jacobs 1995). FORR actively encourages the use of multiple learning methods, multiple representations, and multiple decision rationales. Briefly, FORR's reasoning method combines multiple rationales for problem solving. Although any one of these procedures could be used to search for a solution alone, the expectation is that together they will search more efficiently and effectively. Prior to execution, the user partitions the rationales, separating the correct ones from the heuristics. During search, at each decision point, FORR uses the rationales to choose an action. If no correct rationale can identify an action in the current state, then all the heuristics are consulted together and a combination of their suggestions becomes the decision.

One constructs a FORR-based program for a particular set of related problem classes, such as game playing or path finding. This construction requires programming that specializes FORR with knowledge, including what to learn about any problem class, how and when to learn it, and what rationales generally underlie good decisions over the set of problem classes. It is the programmer's responsibility to specify whether each rationale is expected to be correct or merely heuristic. ACE is a FORR-based program for constraint solving.

A FORR-based program learns when it attempts to solve a problem, or when it observes an external expert solve one. A solution attempt is a sequence of decisions. For example, ACE alternately selects a variable and assigns a value to it. Thus an ACE solution to the problem in Figure 1 might be "choose A, assign A = 2, choose B, assign B = 2, choose C, assign C = 1, choose D, D= 2."

FORR learns to combine rationales to improve problem solving. FORR also acquires *useful knowledge* (probably accurate and possibly reusable data) for each problem class it encounters. A FORR-based program solves easy problems quickly; hard problems take longer. A FORR-based program does not make obvious errors. It can learn new decision-making rationales on its own, and readily incorporate them into its reasoning structure. When a FORR-based program recognizes a situation common to problems in its classes, it uses one or more rationales directly responsive to that situation to produce a (possibly ordered) set of relevant decisions (Epstein 1998). Finally, a FORR-based program distinguishes, according to the programmer's specification, between correct rationales and merely reliable rationales by the way it organizes its rationales.

4.1. Advisors and the Decision Hierarchy

A class-independent, decision-making rationale in FORR is called an *Advisor*. The role of an Advisor is to support or oppose any number of current legal actions. A FORR-based program, such as ACE, makes decisions based upon its Advisors' comments. FORR controls and measures the learning process. Both the formulation of Advisors and their organization rely heavily on knowledge about the set of problem classes.

Each Advisor is represented as a time-limited (and therefore limitedly rational) procedure. The input to each Advisor is the same: all learned useful knowledge, the current state, and some set of legal actions in that state. The output of each Advisor is also uniform; it is a set of *comments*, triples of the form:

$$< \text{strength, action, Advisor} >$$

where the *strength* of a comment is an integer in [0,10] that indicates the Advisor's degree of support (above 5) or opposition (below 5). The procedures themselves, however, are not bound to any particular regulations, and they may rely on special-purpose knowledge representations. To *consult* an Advisor is to solicit comments from it, that is, to execute it.

FORR treats a problem solution as a sequence of decisions from one state to the next. For a solved problem, the first state in the sequence describes the problem, and the last state is a desired solution. This sequence of decisions is generated from the Advisors' comments.

FORR-based decision making is summarized in Figure 3. There are three *tiers* of Advisors, subsets categorized by their trustworthiness and their ability to suggest individual decisions or sets of decisions. Tier 1 Advisors and tier 3 Advisors comment only on individual actions (in ACE, a variable selection or a value selection); tier 2 Advisors comment on *subgoal approaches*, sets of actions. The programmer initially partitions Advisors into tiers. There is no requirement that the Advisors be independent, although those in earlier tiers clearly take precedence over those in later tiers.

```
Until the problem is solved or proved impossible
   Execute the action returned by Selection(current state, Advisors)

Selection (current state, Advisors)
Candidates ← all legal actions from the current state
-------------------------------------------------------------------
Tier 1:For each Advisor A in the ordered tier 1
      Comments ← comments of A on Candidates
      If A has absolute authority & Comments recommend actions R
         then select any action in R and return it
      If A can veto & Comments veto actions V & non-empty(Candidates−V)
         then Candidates ← Candidates − V
      If |Candidates| = 1
         then return the single legal action
         else continue
-------------------------------------------------------------------
Tier 2:Unless there is a current subgoal approach
         For each Advisor A in tier 2
            Generate approaches
            If approaches are generated
               then select one and return it
               else continue
-------------------------------------------------------------------
Tier 3:Comments ← For each Advisor A in the unordered tier 3
                     collect comments of A on Candidates into Comments
      Return best action based on voting(Comments)
```

FIGURE 3. ACE's decision-making algorithm.

Tier 1 Advisors specify a single action, and are expected to be correct and at least as fast as those in other tiers. Tier 1 Advisors permit a FORR-based program to make easy decisions quickly. The programmer can endow a tier 1 Advisor with absolute authority or with veto power. With *absolute authority,* whatever action the Advisor mandates in a comment is selected and executed, and no subsequent Advisor in any tier is consulted on that iteration. For example, ACE's *Victory* has absolute authority; it forces the selection of any remaining value when the chosen variable is the last unassigned one. If a tier 1 Advisor has *veto power*, the actions it opposes are eliminated from further consideration by subsequent Advisors. For example, ACE's *Degree Zero* has veto power; it implements the rationale "if a variable has no neighbors in the constraint graph, do not select it." Such a variable can, of course, be postponed, because it will not conflict with any subsequent assignments. Thus Degree Zero is correct, and it can be implemented to run quickly. Tier 1 Advisors are consulted in order of relative importance, as pre-specified by the programmer. Victory is always first in the ordering for tier 1, because there is no point in further computation once only a single unassigned variable remains.

Tier 1 Advisors are consulted in sequence until either a single decision is selected by an Advisor with absolute authority, or until vetoes reduce the set of candidate actions to a single action. In either event, the selected action is executed to produce the next state.

If tier 1 has not made a decision and there is no current subgoal approach, all remaining (not vetoed) actions are ordinarily forwarded to the Advisors in tier 2, which attempt to generate a subgoal approach. Subgoals for constraint solving, however, are outside the scope of this paper. Instead, the flow of control for ACE is pictured in Figure 4, and any remaining actions after tier 1 are forwarded to tier 3, where all the Advisors comment in parallel.

4.2. Tier 3 Advisors and Voting

Tier 3 Advisors are heuristics that specify a single action, without absolute authority, veto power, or any guarantee of correctness. To select an action in tier 3, a FORR-based program

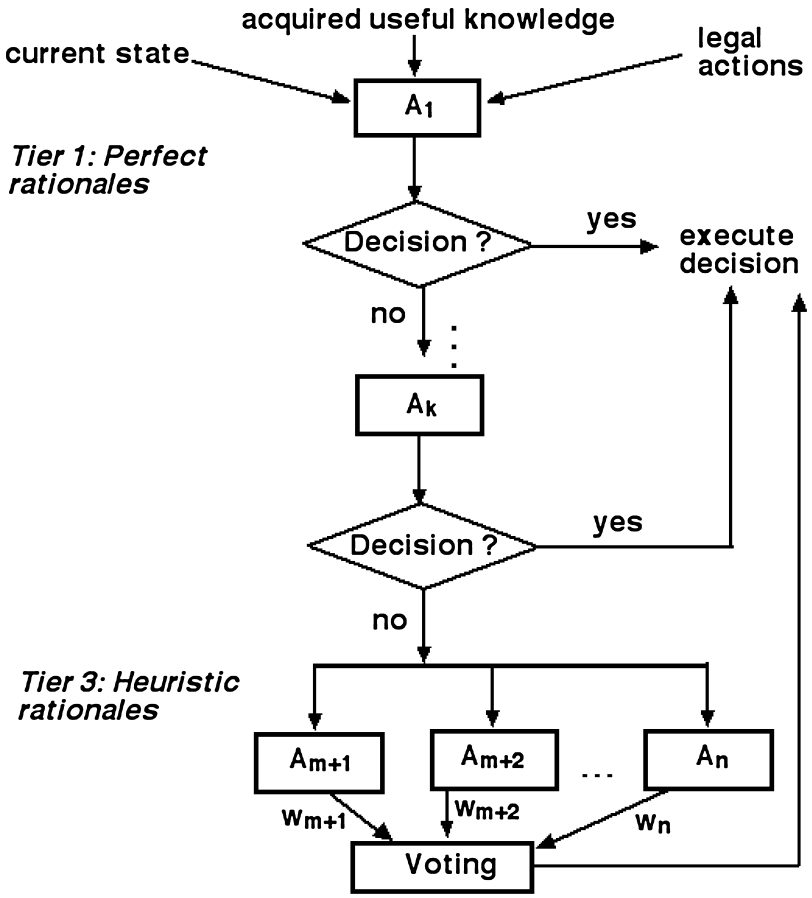


FIGURE 4. ACE's decision hierarchy.

combines the comments of its tier 3 Advisors. Each Advisor can comment on any number of actions. Unlike tier 1, where Advisors are consulted one at a time, in tier 3 all Advisors are consulted at once, and a weighted combination of their comments produces a decision. (The origin of the weights is described in the next section.) If no single action is deemed best, one from among the best is chosen with a method specified by the user (i.e., either lexical or random tie-breaking).

ACE's basic (i.e., non-learned) tier 3 Advisors derive from a set of basic properties called *metrics*. Each metric gives rise to a pair of *dual* heuristics, one of which seeks to minimize the metric, and the other seeks to maximize it. A metric returns a number for each possible action, for example, the degree of a variable or the number of times a value has already been applied. Some metrics are gleaned from the literature, some are common CSP lore, and others are naively hypothesized. A *static* metric is computed once, before problem solving begins; a *dynamic* metric is recomputed periodically during search. For example, one traditional metric for variable selection is the static *degree* of a variable, the number of neighbors it has in the original constraint graph. The heuristic *Maximize Degree* supports the selection of unvalued variables in decreasing degree order. For example, in Figure 1, Maximize Degree would prefer variables A and D, while Minimize Degree would prefer B

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

FIGURE 5. The voting computation for tier 3.

and C. Although Maximize Degree is popular among CSP solvers, ACE also implements its dual, *Minimize Degree*, which supports the selection of unvalued variables in increasing degree order. Another example of a metric, this time a naïve, dynamic one for value selection of an already-chosen variable, is *common value*, the number of variables already assigned this value. *Minimize Common Value* supports the selection of values less frequently in use in the partial assignment; *Maximize Common Value* is its dual. A full list of ACE's Advisors and their rationales appears in the Appendix.

Each Advisor has a learned weight and a discount factor. The *discount factor* serves to introduce each Advisor gradually into the decision process. Weights are discounted until an Advisor has commented 10 times during learning, with the expectation that by then its weight will be representative of its accuracy.

ACE makes decisions in tier 3 through *voting*, a process that combines the comments of its Advisors to determine the action with the greatest support. (See Figure 5.) Voting multiplies the strength of the opinion of each Advisor on each action by the Advisor's weight and the Advisor's discount factor. These weighted strengths are summed across Advisors for each action. The winner of the vote is the action with the highest support. As they arise, learned new heuristics participate in the decision process in the same manner—they enter with a discount factor and either they vote individually, or the summary heuristic votes on their behalf.

Clearly, to participate in voting, an Advisor must be consulted and then comment. A *benchmark* is a nonvoting, baseline procedure, which is presented with the same actions as the Advisors it gauges, and comments with randomly generated strength on r randomly chosen actions $(0.5)^r$% of the time. A benchmark models how well random comments would do in the same decision situations. (Commenting on only r of the choices simulates the ability of an Advisor to discriminate among them.) Because only two kinds of weighted decisions arise in ACE (tier 3 variable selection and value selection), there are only two benchmarks: one for variable selection and one for value selection. Although all Advisors are consulted during learning, only those that have earned a weight greater than that of their respective benchmark are consulted during testing. Q2

5. LEARNING MECHANISMS

Primarily, what ACE learns is how to combine its Advisors to solve problems in a particular class. In support of this, ACE learns the weights used in voting for its tier 3 Advisors. These weights serve a variety of purposes: they filter out inaccurate heuristics before testing, they identify accurate learned heuristics and their relative accuracy, they support the reorganization of tier 3, and they determine how to replace ACE's decision-making mechanism in Figure 3 with a more simplistic computation.

5.1. Weight Learning

Even though a set of heuristics is deemed appropriate, they may not all be of equal significance or reliability within a particular problem class. Weight learning fits a FORR-based application to correct decisions, learning to what extent each tier 3 Advisor reflects expertise in a particular problem class. The comments of an often-correct Advisor gradually have more influence during voting, while those of an often-incorrect or occasionally egregiously incorrect Advisor are soon overwhelmed.

As observed in Section 2, regardless of κ value, the variation in problem difficulty *within* a class is notoriously heavy-tailed. Therefore, among any set of randomly generated problems within a single class there will be some that are very easy (readily solvable by any method), and others that are extremely difficult. The difficult problems will be frequent, and success on them will tax any algorithm that is merely ordinarily successful on the rest of the class. Thus the reputed uniformity of a problem class is deceptive, and a program that intends to learn within a problem class will be confronted, not by noise, but by inherent, deceptive variations in problems within the same class, both during learning and during testing.

The algorithm we describe here, *DWL* (Digression-based Weight Learning), learns problem-class-specific weights for tier 3 CSP Advisors. DWL is crafted for problems where both errors and their cost in search nodes can be identified. DWL is intended to balance a set of Advisors, some of which may not be present initially but arise during learning. DWL is also deliberately tailored to manage problems whose difficulty lies on the hard tail of the difficulty distribution. It therefore somewhat undervalues strong performance on (presumably easy) problems, with the presumption that harder problems will soon appear. DWL learns only from solved problems, and only after a solution.

The premise behind weight learning in FORR is that the past reliability of an Advisor is predictive of its future reliability. DWL 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 examples*, pairs of the form $<s, d>$ where s is a state, d is the decision ACE made in tier 3 for that state, and more than one possible action remained after tier 1. The intuition behind DWL is suggested by Figure 6, which diagrams the steps in search to a solution for the problem of Figure 1 using the variable ordering (B A C D), the value ordering (1 2), and no inference method. The solid path in Figure 6 is the *underlying perfect search path*; it includes exactly $2n$ correct decision steps, represented along it as black circles. Those decisions should be reinforced. The remainder of the search consists of *digressions*, subtrees whose roots (represented as white circles in Figure 6) are eventually retracted as bad value assignments. A decision at the root of a digression is an error that produces an over-constrained subproblem; that decision should be discouraged. Because digressions can contain millions of steps, and because learning occurs after (not during) the solution of each problem, nonroot steps in digressions (represented as gray circles in Figure 6) are not retained or learned from; only the size of a digression (in number of steps) is recorded. Finally, a value selection only serves as a training example if the forward degree of the relevant variable is not zero.

For DWL, the quality of a decision is determined by the role that it played in each search for a solution. Good decisions lie on the underlying perfect search path; bad decisions are either value assignments that immediately preceded a digression, or variable selections that led to those value assignments. DWL rewards Advisors that support good decisions or oppose bad decisions with weight increments, and penalizes Advisors that oppose good decisions or support bad decisions with weight decrements. Penalties are assessed in proportion to the size of the digression; rewards are provided in proportion to the size of the search tree relative to other problems in the same class. Furthermore, DWL discounts both rewards and

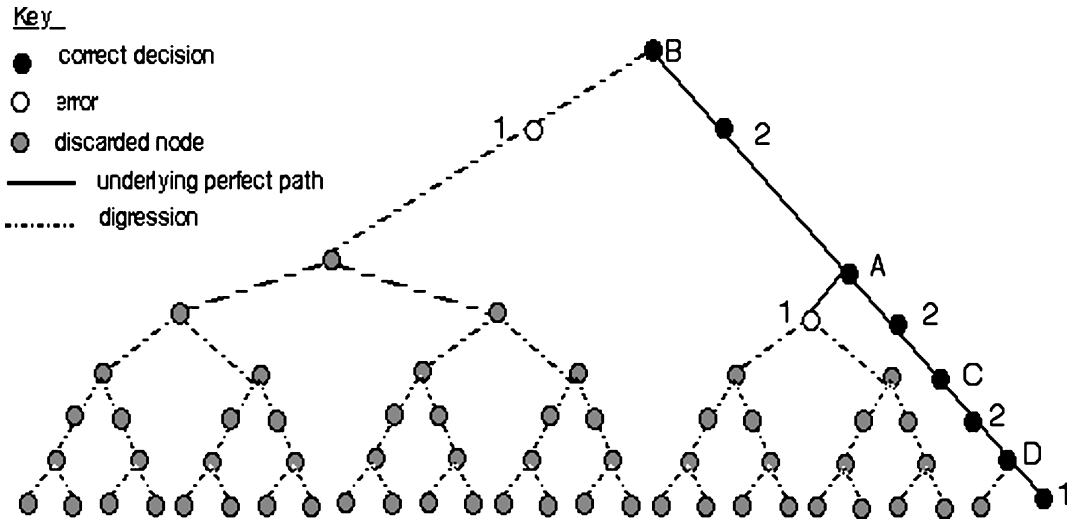


FIGURE 6. A search tree for a CSP. The eventual solution is A = 2, B = 2, C = 2, D= 1.

penalties in proportion to the number of choices each Advisor ranks as best, in response to the Advisor's inability to discriminate among them. A detailed discussion of the issues underlying unsupervised weight learning for CSPs, along with ablation experiments highlighting the role of each facet of the algorithm, are available in (Epstein, Wallace, and Freuder, in review).

DWL learns weight w_i for Advisor i with the algorithm in Figure 7. Under DWL, all tier 3 Advisors and their benchmarks begin as equally significant, with weights of 0.05 at the beginning of each run in an experiment. Training examples are collected during each problem. If ACE solves the problem, DWL gauges the problem difficulty by the number of steps required to solve it, and monitors data on all problems solved thus far, assessing rewards and penalties to Advisors for each training example. The comments of an Advisor on a training example and their relative strengths, determine whether or not the Advisor is correct there. An Advisor is correct on $<s, d>$ if and only if it prefers d over any other legal action in s. A comment with strength in (5, 10] is interpreted as support, and one with strength in [0, 5) is interpreted as opposition. The absence of a comment on a particular action is interpreted as a comment with strength 5.

```
For each training example <s,d> with state s and decision d
 For each Advisor A_i that produces comments c_i on s
   d_i ← d_i + 1
   If the next state s' was not the root of a digression
      then if c_i supports d,
      then increase w_i
      else decrease w_i
    else if c_i supports d,
      then decrease w_i
      else increase w_i
```

FIGURE 7. A high-level version of DWL.

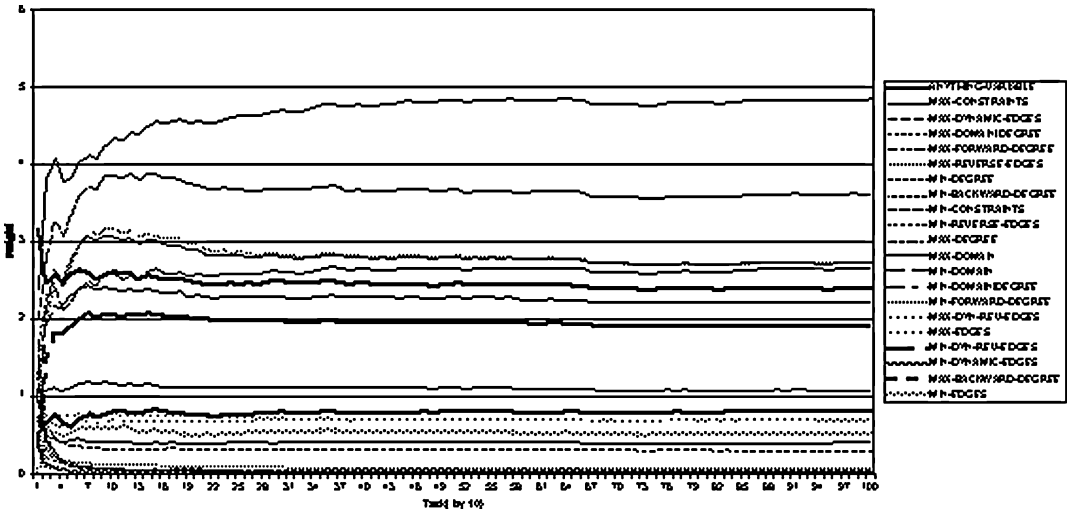


FIGURE 8. The weights of ACE's variable-ordering heuristics and their benchmark (heavy line) across as ACE solves 1000 problems from <30, 8, 0.1, 0.5>. Similar settling occurs with every CSP class we have studied thus far.

An interesting property of DWL is that corrections become smaller, relative to the overall weight, as learning progresses. Consider, for example, an Advisor that gets the first training example correct, and is wrong on the second. (For clarity here, we ignore the discount factor, and assume uniform rewards of size 10, without penalties.) After the first example, the Advisor's weight is 10 times the number of examples on which it gave correct advice, divided by the number of examples on which it gave advice: $10/1 = 10$. The error on the second example, however, increases the denominator to 2, without changing the numerator, so the new weight is $10/2 = 5$, a reduction of 50%. Now consider an Advisor that gets the first 100 training examples correct, and is wrong on the 101st. After the first 100 examples, the Advisor's weight is $1000/100 = 10$, and after the error it is $1000/101 = 9.90$, a reduction of 1%. Figure 8 graphs the stabilization of the weights of ACE's variable-ordering heuristics from a single run where it learned on 1000 problems in <30, 8, 0.1, 0.5>. (We note, however, that there may be more than one set of values to which ACE might settle, that is, the weight set is not necessarily unique.)

5.2. Weight Stages

Even a heuristic-judged valuable within a particular problem class might not be equally good at every search depth. It may, for example, be most useful only at the top of the search tree. *Stage breaks* partition problem-solving decisions during an experiment. For example, *20–80 stage breaks* in ACE define an *early stage* where no more than 20% of the variables are bound, a *middle stage* where between 20% and 80% of the variables are bound, and a *late stage* where more than 80% of the variables are bound. In FORR, weights are learned for each stage separately. The user may specify a fixed number of equal-size stage breaks (no more than n in ACE), a monotonically increasing list of percentages (e.g., 20, 80), or a monotonically increasing list of number of steps (e.g., 2, 4, 6, 10).

The correct number of stage breaks and their placement are thus far set empirically, and appear to be dependent on the problem class. Too many stages (e.g., 30 for <30, 8, 0.26,

0.66>) overfits the data and degrades performance; indeed, a single stage often suffices. In some classes (e.g., graph coloring), however, 20–80 stage breaks seem prescient. (Those particular values were actually "inherited" from a FORR-based game-playing program that used them to isolate the opening and the endgame from the middle game.) The results in Sections 6.3 and 6.4 were achieved with 20–80 stage breaks. All the other results reported here were produced with a single stage (i.e., no stage breaks).

5.3. Learning New Heuristics

FORR supports learning new heuristics from a regular language, each of whose expressions can be interpreted as an Advisor. (Further details appear in Section 7.2.) To learn new heuristics within such a language, FORR monitors how each possible expression would perform if it participated in problem solving. First, FORR generates the legal expressions for the language with the grammar. Then, after each learning problem, each expression is given the opportunity to comment on each training example, as if the expression were an active heuristic. For each expression, FORR tallies how often an expression's comments discriminate among the actions, and how often those comments are correct. Initially, these expressions are not permitted to participate in ACE's actual decisions; they are merely monitored to estimate their accuracy.

During learning, the status of each expression in a language is either potential, active, spawned, or inactive. Initially, all expressions are *potential*, that is, monitored for possible inclusion in decision making. After every p learning problems ($p = 10$ here), which may have produced hundreds of training examples, FORR reevaluates each expression's status. Those that fail to discriminate or never comment are eventually made *inactive*, to speed subsequent computation. Potential expressions that achieve a weight of at least 8.5 (out of 10) are promoted to *active* status.

Eventually, accurate expressions are used collectively during decision making, and highly accurate ones are used individually. Active expressions provide input to their language's *summary heuristic*, which combines their comments to structure its own. Once any expressions become active, the summary heuristic participates as an Advisor in ACE's decisions. If an active expression achieves a weight of at least 9.5, it is *spawned*, that is, transformed into an individual heuristic that participates in ACE's decisions. A spawned expression no longer participates in the summary heuristic's computation. Both the summary heuristic and any learned heuristics for an Advisor language are subject to the discount factor, so that they enter the decision process gradually. This permits ACE to maintain its performance level as it introduces new Advisors.

6. RESULTS

Experiments with ACE have a uniform design. In a *run*, we train the program, then turn learning off and test the program. To minimize the chance that ACE was merely fortunate enough to encounter "good" problems in the class from which it learned, we average performance for each experiment across 10 runs. On each run, ACE learns on a different set p_l of problems (10 p_l learning problems in all). On every run, however, ACE confronts the same set p_t of testing problems (distinct from all the learning problems), to judge the program's performance after different learning experiences against the same testing set. Because ACE could encounter a very difficult problem at any time during learning, we limit computation on any individual learning problem to 10,000 *steps* (total number of variable selections, value selections, and retractions). During testing, however, ACE is required to solve every problem.

There is no overlap between the learning problems and the testing problems. The user designs an experiment. All experiments reported here used MAC3 maintained arc consistency (Mackworth 1977) as an inference method, chronological backtracking, random tie-breaking during learning, and lexical tie-breaking during testing.

This section recounts a variety of experiments with ACE. Section 6.1 shows that ACE's performance improves with its experience, that is, that it learns. Section 6.2 shows that it produces distinct characterizations for different classes of CSPs. Section 6.3 demonstrates that ACE can learn heuristic combinations related to those an expert might choose. ACE's rediscovery of the Brélaz heuristic, and Section 6.4 describes how ACE discovered a new heuristic. Section 6.5 explains how ACE determines when it has learned enough, and Section 6.6 explains how ACE can reformulate its decision-making structure.

6.1. Learning to Solve Hard Problems

Our first experiment demonstrates that ACE *learns* to solve hard problems, (rather than having been tuned to do so in advance), and that the training set, despite the variety of problem difficulty, does not impact the ultimate testing performance. This experiment uses the class <30, 8, 0.26, 0.66>, which lies at the phase transition and is therefore predicted to be particularly difficult for CSP solvers in general. Throughout this discussion, any cited difference is statistically significant at the 95% confidence level.

- *ACE learns to do less work.* The average number of constraint checks during learning was reduced by 72% during testing. The greatest number of constraint checks on any individual testing problem was 418,675, versus 8,791,577 during learning.
- *ACE learns to move more incisively through the partial assignment space.* The average number of steps during learning was reduced by 53%. The maximum number of steps during any testing problem was 532, versus 9357 during learning.

In this experiment, ACE began with 40 general, constraint-solving heuristics, listed in the Appendix. ACE is faster on testing problems in part because, during testing it does not devote any computation time to the 22 low-weight heuristics that do not meet the cutoff after learning. The remainder of the speedup is primarily attributable to fewer errors; indeed, the time spent on retraction was reduced by 76% during testing.

We used $p_l = 80$ learning problems and $p_t = 50$ testing problems. If ACE is *learning* to solve these problems, performance during testing should be markedly better than performance during learning, and that indeed proves to be the case. The first two sets of data in Table 1 compare ACE's performance during learning and testing. ACE solved most of its learning problems (subject to a 10,000-step limit) and was required to solve all of its testing problems. To display the impact of the heavy tail, both average and median values are included here. Several points are noteworthy:

- *ACE learns to solve problems faster.* Despite the requirement that ACE solve all its testing problems, the average time spent on a testing problem is 85% less than the time spent on a learning problem. The longest time spent on any individual testing problem was 32.49 seconds, versus 1093.93 seconds during learning.
- *ACE learns to make fewer mistakes.* The average number of retractions during learning was reduced by 79% during testing. The most retractions on any individual testing problem were 472, versus 9947 during learning.

TABLE 1. ACE's Performance on a Set of Hard Problems <30, 8, 0.26, 0.66>, Compared With an Off-The-Shelf CSP Heuristic

Program		Solved	Time	Checks	Nodes
ACE during learning	Mean	97.875%	42.83	401565.03	508.47
	Median		8.66	90208.00	105.00
ACE during testing	Mean	100.000%	6.62	113292.60	131.62
	Median		5.41	94091.00	111.00
Min Domain	Mean	100.000%	7.00	286708.53	365.84
	Median		4.92	200510.00	249.00

Time is in CPU seconds. The other performance metrics are defined in the text. Median values appear below averages. Similar behavior occurs with all but the easiest CSP classes we have studied thus far.

Median values for retractions, constraint checks and steps actually rose during testing, because ACE was forced to solve every problem, including the most difficult ones. We therefore also present a second argument that learning is taking place: the effort devoted to solving the first 10 learning problems in each run, compared to the second 10, the third 10, and so on. This is displayed in Figure 9 measured by steps, constraints checks, and time. Note, on every performance metric, the sharp improvement between the first 10 tasks and the second 10, followed by deterioration during the third set of 10, and then continued improvement and apparent performance stabilization. Observe, too, in Figure 9, how the median measures are less subject to the heavy tails and do not exhibit as much difficulty with the third set of 10 problems.

In this experiment, ACE learned weights in [0,10] for tier 3 Advisors. Although CSP research suggests which member of each pair is correct, we left it to ACE to detect the valid member; those results from this experiment appear in Table 2, where the cutoffs for using heuristics in Table 2 are the benchmarks' weights. Among the variable-ordering heuristics, the two widely acknowledged in the constraint literature were learned as significant on every run: *Maximize Degree* (maximize the degree of the variable in the original constraint graph), and *Minimize Domain/Degree* (minimize the ratio of the dynamic domain size to the degree in the original constraint graph). Observe that the heuristics with the highest weights were learned regularly (i.e., on either 9 or 10 runs). Only one metric was learned with its dual, the only metric not drawn from the CSP literature.

Inspection of the runs themselves indicates that the variation in difficulty *within* a CSP class is indeed an issue. On two of the 10 runs, ACE got off to a bad start—it encountered (what we believe were some relatively easy) problems on which almost any heuristic, or even random selection, would have worked, and then reinforced whichever heuristics matched its solution trace. Indeed, for a while on one of these troublesome runs, ACE's weights indicated that one should minimize degree and maximize domain/degree ratio, just the opposite of the common wisdom. Nonetheless, by the 30th problem, ACE had recovered, and had begun to identify the correct heuristics. In 7 out of the 10 runs, ACE also solved 1 or 2 of the 50 testing problems without any retractions at all; that is, it made flawless decisions. These problems presumably came from the expected "easy" tail of the distribution.

ACE's errors are carefully monitored. During testing, the average error occurred when 3.98 variables were bound. No error on *any* testing problems occurred after 11 variables had been assigned values. This observation will be used to formulate the experiment in Section 6.6.

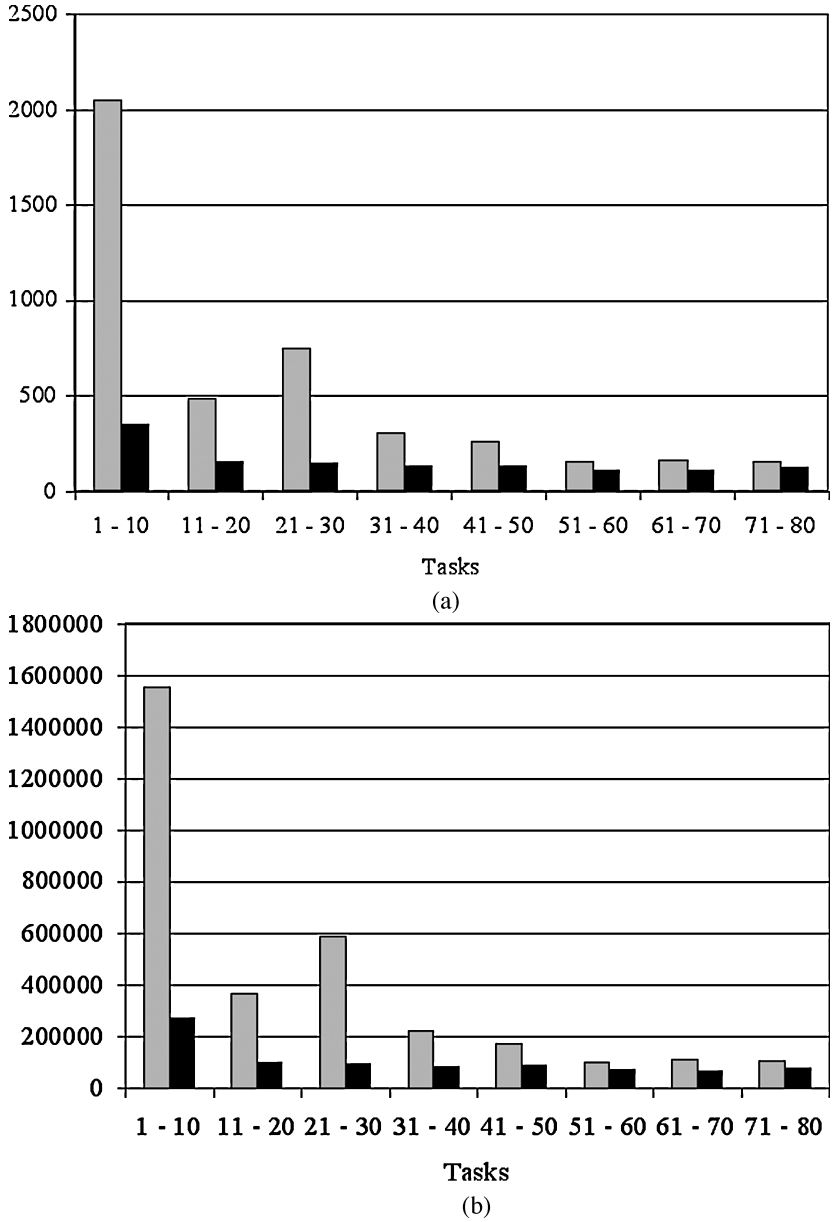
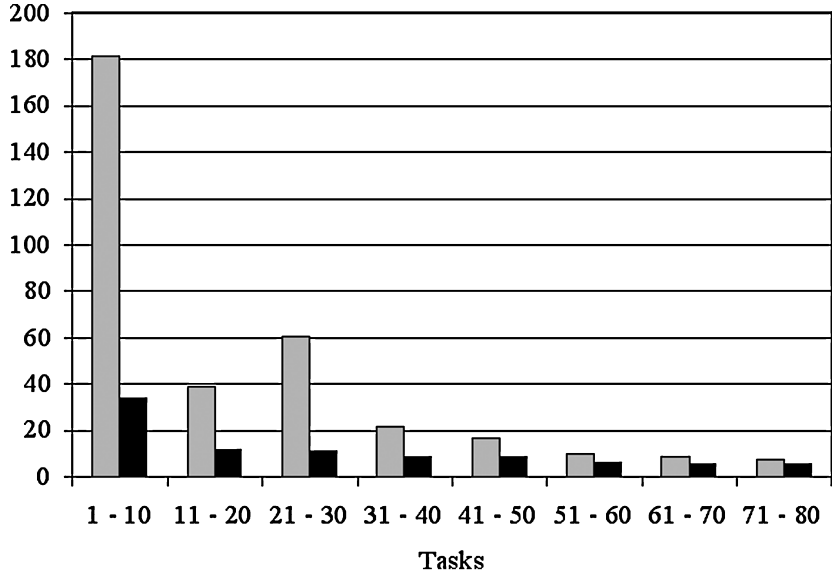


FIGURE 9. Across 10 different training experiences, (a) the number of steps, (b) constraint checks, and (c) time ACE required on the first ten problems, the next ten, and so on, in a set of 80 from <30, 8, 0.26, 0.66>. Lighter bars represent average values; darker bars represent medians. Similar behavior occurs with all but the easiest CSP classes we have studied thus far.

It would not be sufficient for ACE merely to identify accurate heuristics and then combine them without weights. We demonstrate this with two voting variations on the same testing set. The first set of data in Table 3 shows how ACE does when all the tier 3 heuristics vote without weights (*no learning*). The second set in Table 3 shows the performance of only the 16 heuristics ACE found valuable (from Table 2), again without weights (*weightless*). In both



(c)

FIGURE 9. (Continued)

cases a step limit of 10,000 was enforced, and there were 10 runs. The no-learning version failed to solve 5% of the problems within 10,000 steps; ACE and the weightless version of ACE solved all the testing problems. Because ACE's performance during testing is far better than during learning, we conclude that the program learns to solve hard problems. Because this performance is consistent on the same testing set after training on 10 different sets, we conclude that the training set does not inordinately influence ACE's learning. Finally, the subset of best heuristics identified by ACE clearly outperforms the full set, and ACE's learned weights clearly improve the decisions the best heuristics make.

6.2. Learning to Characterize a Problem Class

Our next experiments explore ACE's ability to characterize a problem class by the Advisors it identifies as accurate across runs. Recall that an Advisor is used during testing only if its weight surpasses that of its benchmark. Such an Advisor produces comments that are more reliable than random comments. If ACE uses an Advisor during testing on every (or nearly every, say 9 out of 10) run, we say that it is *characteristic* of the problem class used during the experiment. The set of such Advisors may be said to characterize a problem class; we call it the *weight profile* for a problem class. In general, these weight profiles appear to be consistent, that is, they are independent of the problems used during learning.

Table 4 lists ACE's weight profiles for several different problem classes, as minimizing or maximizing particular variable-ordering tier 3 metric. The problem classes profiled are random problems in <30, 8, 0.1, 0.5> and <30, 8, 0.26, 0.66>, geometric problems in <20, 10, 0.6, 0.3> and quasigroups of order 10 with 60 holes. Table 4 shows that random and geometric problems have roughly similar profiles, although there are some discrepancies. Quasigroup problems, on the other hand, have distinct profiles; in particular, we found that some Advisors that represent antiheuristics for random problems are favored over their opposites when quasigroup problems are tested. In this case, it is clear that profiles for one

TABLE 2. Ordered by Weight Within Type, Weights in [0, 10] that ACE Learned for CSP Heuristics and Then used During Testing on the Hard Class of Problems, <30, 8, 0.26, 0.66>

Heuristics	Weight	Times Learned
Value-ordering		
Maximize product domain value	1.23	10
Maximize options value	1.04	10
Minimize static conflicts value	1.04	10
Minimize domain score value	1.02	10
Maximize secondary pairs value	0.98	10
Maximize common value	0.97	5
Maximize small domain value	0.97	5
Minimize common value	0.93	2
Maximize secondary value	0.85	2
Maximize weighted domain score value	0.75	1
Variable-ordering		
Maximize value pairs	3.44	9
Minimize domain/degree	2.98	10
Maximize dynamic connected edges	2.84	9
Maximize static degree	2.55	10
Maximize static connected edges	2.22	6
Maximize backward degree	1.80	4

"Times learned" is the number of runs (out of 10) on which the heuristic was recognized as valuable. All heuristics are defined in the Appendix. Cutoffs for value-ordering and variable-ordering heuristics are provided by their respective benchmarks, as discussed in Section 5.1. Similar consistency (albeit for different heuristics) occurs with all but the easiest CSP classes we have studied thus far.

class are not appropriate for the other. This has also been verified with a non-ACE constraint solver.

6.3. Learning Heuristic Combinations (Those a Novice Might not Know)

Our next experiments demonstrate ACE's ability to discover empirically heuristic combinations that would be related to those known by problem-domain experts or CSP experts, but likely unknown by novices. Recall that ACE has dual pairs of heuristics, one of which maximizes a metric while the other minimizes it. Using the wrong member of such a pair (the *antiheuristic*) typically makes search intractable on problems that the heuristic itself is able to solve. By including the antiheuristics, we challenge ACE to learn which heuristics are correct, and in what proportions. We ran two experiments, with forward checking as the inference method, in which ACE had only variable-ordering heuristics (i.e., no value-ordering heuristics) and the learning problems were a class of four colorable graphs on $n = 20$ variables, with $d = 0.1$. Again ACE did 10 runs, learning on 80 problems (a new set for each run) and testing on 50 additional problems, with the same testing set on each run. In both experiments, ACE solved all the problems on every run, and solved all the testing problems without retraction.

TABLE 3. The Impact of Weight Learning Is Demonstrated Here by Comparing the Performance of Several Programs on Problems in <30, 8, 0.26, 0.66>: ACE and Two Ablated Versions of ACE

Program		Solved	Time	Checks	Nodes
All ACE heuristics without weights	Mean	95.00%	144.48	1355083.10	1664.57
	Median		47.63	439726.00	547.00
Best ACE heuristics without weights	Mean	100.00%	7.98	116054.63	133.84
	Median		6.82	108017.00	128.00
ACE (best heuristics, weighted)	Mean	100.00%	6.62	113292.60	131.62
	Median		5.41	94091.00	111.00

Time is in CPU seconds. Other metrics are defined in the text. Median values appear below averages. Similar behavior occurs with all but the easiest CSP classes we have studied thus far.

In our first coloring experiment, ACE was given only *Later*, a heuristic that eliminates from consideration any variable whose dynamic domain size is larger than its degree (unless they all are), and four pairs of dual variable-ordering heuristics for static degree size, dynamic domain size, *forward degree* (number of as-yet-unvalued neighbors), and *backward degree* (number of valued neighbors). The combination of Later and the ability-to-learn weights proved powerful here. On every run, three variable-ordering heuristics were selected: minimize dynamic domain size, maximize degree, and maximize forward degree. (On seven runs a single heuristic, maximize backward degree, was also selected.) These heuristics resemble the well-known Brélaz heuristic for graph coloring: minimize the domain size and, in the event of a tie, maximize the degree. Moreover, the combination of the two is consistent with the evidence presented in Bessière and Régin (1996) that minimizing domain size/degree is a superior CSP-ordering heuristic to either minimizing domain size or maximizing degree

TABLE 4. Weight Profiles Learned by ACE for a Sample of Four Different Problem Classes

Metric	Random <30, 8, 0.1, .5>	Random <30, 8, 0.26, 0.66>	Geometric <20, 10, 0.6, 0.3>	Quasigroups 10×10, 60 holes
Degree	Max 10	Max 8	Max 9	–
Domain	–	–	(Min 1)	(Min 3)
Forward degree	–	–	–	Min 8
Backward degree	–	–	(Max 3)	Max 8
Domain/degree	Min 9	Min 10	Min 10	(Min 3)
Value pairs	Max 10	Max 10	(Max 2)	Min 7
Edges	Max 10	(Max 4)	Max 9	–
Dynamic connected edges	Max 10	(Max 6)	–	Min 8
Dynamic less-connected edges	–	–	–	Max 8 Min 2

All metrics are defined in the Appendix. Entries indicate whether the metric was judged accurate as a maximum or a minimum, and on how many runs out of 10. Entries in parentheses are not sufficiently frequent to be characteristic of the problem class. (Unlike those of Table 2, these experiments used 20–80 stage breaks.) Distinct weight profiles conform to the CSP literature and were expected.

alone. Given the relatively recent vintage of this insight, its "rediscovery" by FORR is impressive. The success of Maximize Backward Degree may well reflect its correlation with both Minimize Domain (the domain will be reduced for each differently colored neighbor) and Maximize Degree.

In our second coloring experiment, ACE was given Later plus nine pairs of variable-ordering heuristics. This time, six heuristics were selected on every run: maximize constraints, maximize degree, maximize dynamic edges, maximize forward degree, maximize reverse edges, and minimize the ratio of domain to degree. (On no more than two runs, three others were mentioned, but with lower weights.) As a result, ACE required more time to make decisions and there was no appreciable difference in what was already very good performance. In Section 6.6 we suggest how ACE can recover from this surfeit of knowledge. In both cases, even when forced to cope with a larger initial set of heuristics, ACE' has learned to discard the antiheuristics and to emphasize what experts in this field have proposed and tested: minimize the domain size and use heuristics related to maximizing the degree. Q3

6.4. Learning Valid New Heuristics (An Expert Might not Know)

This section describes ACE's ability to learn and validate possibly hitherto unknown heuristics. There is no reason to believe that ACE's input heuristics are in any way exhaustive. A particularly powerful one is the ratio of two of the others (minimize the ratio of the dynamic domain size to the static degree, mDD). This led us to wonder about the efficacy of other arithmetic combinations of metrics. We therefore formulated an *arithmetic variable language* in which a learned heuristic's metric would be a function of a pair of variable selection metrics. (Further details on such languages appear in Section 7.2.) Results in the prior sections did not include learning heuristics; there, ACE relied only on its initial set.

First we had ACE learn on the class <30, 8, 0.1, 0.5> and tested it there. Next, we removed the pair of heuristics on the Domain/Degree metric from ACE (call that version ACE–) and reran the experiment. There was no statistically significant difference in performance between ACE and ACE– along any metric. This suggested that mDD might be unnecessary, despite its high weight in the first experiment.

Next, we retested ACE–, this time adding the ability to learn new heuristics with a language capable of expressing the Domain/Degree metric and using the 20–80 stage breaks described in Section 5.2. This language provides only heuristics not included in ACE–. Again there was no change in performance. Moreover, although the expression for mDD is part of the arithmetic variable language, it never became active even once in 10 runs. (Recall from Section 5.3 that only when an expression has a high estimated weight does it become active. Any number of expressions can become active if their estimated weights are high enough.) Instead, exactly one expression became active on every run: "maximize the product of degree and forward degree when no more than 20% of the variables have been bound" (PDFD). With PDFD, ACE– with the ability to learn heuristics performed just as well as ACE and as ACE–.

To explore this outstanding, learned heuristic further, we incorporated PDFD into a CSP algorithm coded in a conventional manner, and tested it on reasonably difficult 150-variable problems in <150, 5, 0.05, 0.24>. Our results, in Table 5, are significant for several reasons. They demonstrate that lessons learned with ACE can be transferred to a conventional algorithmic context, that lessons learned on easy problems can be relevant to hard problems, and that our conventional understanding of search-order heuristics may be overly simplistic.

We tested how PDFD would impact three CSP conventional heuristics: mDD, minimize domain size (mD), and mD after first ordering the variables by descending degree. First, we tested each conventional heuristic on <150, 5, 0.05, 0.24>. Then we repeated the same tests, this time replacing each conventional heuristic by PDFD in the top fifth of the search tree. It

TABLE 5. Performance Results in Nodes Searched Per Problem for Three Conventional Heuristics, with and without the Product Heuristic PDFD Learned by ACE

Conventional Heuristic	Alone	Enhanced Heuristic
Minimize domain size (mD)	86,065	3,218
mDD	4,277	3,218
mD after degree preorder	12,602	3,218

mDD is the Minimize Domain/Degree heuristic alone. Data are averaged over 10 runs using code separate from ACE. Problems had 150 variables, domain size 5, density 0.05, and tightness 0.24. Similar behavior occurs on other classes of random CSPs tested.

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 increases slightly 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" (e.g., 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 inference 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 inference. We conclude that ACE is indeed capable of making discoveries of interest to human CSP solvers.

6.5. Knowing When to Stop Learning

Recall that part of the design of an experiment with ACE is to specify p_l, the number of learning problems. Termination of learning can be more elaborate than merely counting the number of problems addressed, however. Instead, the user may specify that learning terminate and testing begin after a specified elapsed time, or once some skill is manifested (e.g., 10 consecutive problems solved well). A more recent innovation with DWL is the ability to halt when learning no longer has an impact on Advisor weights.

Because DWL-calculated weight corrections have a diminishing impact (as observed in Section 5.1), ACE can monitor weight fluctuations, to determine on its own when it should stop learning. We call this *learning to stability*. ACE deems its weights to be *stable* (unlikely to change further) when the standard deviation of every Advisor's weight across a window of the w most recent problems is no more than σ. With learning to stability, the number of learning problems becomes an upper bound, so that ACE can choose to stop the learning phase earlier on any run. Both w and σ are parameterized, and ACE typically reaches stability on problems quite quickly. For example, with $w = 50$ and $\sigma = 0.01$, ACE reaches stability on <30, 8, 0.26, 0.66> after only 125.4 problems.

Although increasing the window size w allows more opportunity for a particularly difficult (or easy) problem to appear, we have observed no significant performance differences

with larger w. As one would expect, more difficult problem classes take longer to stabilize. Nonetheless, ACE knows when it is likely to learn no more. Learning to stability shortens learning time, without impacting performance.

6.6. Learning to Restructure Decision Making

The material in this section addresses ACE's speed during testing. ACE outperforms "minimize the dynamic domain size" on the hard problems of Table 1, as measured by constraint checks, retractions, and nodes. ACE is, however, slower. Inspection indicates that it is actually often faster to decide quickly and retract errors than to make a carefully reasoned judgment. Three different approaches to speed up decision making (and thereby search) are considered here: promotion, pushing, and prioritization. Each of them restructures the decision process of Figure 4.

In other FORR-based applications, an occasional Advisor earned a weight so high that the Advisor appeared correct. *Promotion* moves such a highly weighted tier 3 Advisor into tier 1. In some kinds of problems (e.g., game playing) such a risk is intolerable, because a single error results in failure. In constraint solving, of course, an error can simply be retracted. Nonetheless, empirical evidence from ACE suggests that rarely is the speedup available from faster decisions worth the extended search that results from any promoted Advisor's occasional errors. Furthermore, no ACE Advisor achieves weights (typically around 9.5 in other FORR-based applications) that would make it a candidate for promotion.

The problem with promotion is that it does not consider the problem state. In contrast, *pushing* is a kind of promotion that goes into effect only once the problem appears readily solvable, and then bypasses tier 3 for variable selection. During the learning phase, ACE learns when to push by tracking the deepest error e_d, the greatest number of variables ever bound on any problem when a retraction occurred during learning. Then, only for tier 3 variable selection during testing, and only after the number of bound variables is larger than e_d, ACE consults only a single variable-ordering Advisor that qualifies for use during testing. (Empirical results indicate that value selection should not be pushed, that is, after e_d it no longer matters much where one works in the graph, but what one does there still matters a good deal.)

The folklore of constraint researchers includes a belief (heretofore untested) that once "enough" variables have been assigned values "it doesn't much matter what you do." This simplification is achieved by a tier 1 Advisor called *Pusher*, which comments only during testing and only below e_d. (The latter remains fixed during testing, regardless of subsequent experience.) Say, for example, that $e_d = 17$, that is, that the deepest error during learning was observed when 17 variables were bound. In that case, during testing, for all decisions where at least 18 variables are bound, Pusher will use only one Advisor to make the decision. Thus, after e_d, Pusher effectively promotes a single tier 3 Advisor to tier 1, and thereby avoids the time-consuming computations of all the other accurate tier 3 Advisors. If Pusher is given no Advisor, or its Advisor makes no comments, then Pusher makes lexically ordered decisions after e_d. If Pusher's Advisor has several top-ranked choices, Pusher makes a lexically ordered decision from among them. We have tested Pusher both with rationales well respected in the constraint community (e.g., minimize the domain, minimize the domain/degree ratio) and with variable selection in lexical order.

The data in Table 6 indicate that which Advisor should be used by Pusher is problem class-dependent. On <30, 8, 0.1, 0.5>, Pusher with Minimize Domain reduced the median computation time without any significant change in errors; Pusher with Minimize Domain/Degree and lexically ordered pushing produced nearly identical results. On <30, 8, 0.26, 0.66>, however, only lexical ordering produces any improvement: a small (about

TABLE 6. The Impact of Pushing on Two Problem Classes

Program		Time	Retractions	Checks
<30, 8, 0.1, 0.5>				
ACE	Mean	0.50	3.77	10701.75
	Median	0.47		
ACE + push Minimize Domain	Mean	**0.46**	3.76	10701.01
	Median	0.43		
ACE + push Minimize Domain/Degree	Mean	**0.47**	3.76	10701.15
	Median	0.43		
ACE + push Lexical Order	Mean	**0.46**	3.76	10700.57
	Median	0.43		
<30, 8, 0.26, 0.66>				
ACE	Mean	4.15	103.41	113457.16
	Median	2.94		
ACE + push Minimize Domain	Mean	4.06	103.06	113040.56
	Median	3.83		
ACE + push Minimize Domain/Degree	Mean	4.03	103.37	113615.15
	Median	2.83		
ACE + push Lexical Order	Mean	**3.83**	103.57	113654.27
	Median	**2.74**		

Time is in CPU seconds. Other metrics are defined in the text. Values in bold are statistically significant. When to push and what to push appear to be CSP class-dependent.

10%), statistically significant speedup, that is, it was faster (no more errors were made) to *select a variable without computation.* Inspection indicates that the deepest error is surprisingly early in most classes, but that decisions are still nontrivial there: dynamic domains still have multiple values, and the constraint graph is not a forest. Nonetheless, a simple choice becomes the best.

Finally, *prioritization* partitions those tier 3 Advisors retained for the testing phase into a hierarchy of subsets based on their learned weights, as shown in Figure 10. (Empirical evidence suggests that these subsets should be computed by partitioning the range of the Advisor's weights, rather than be of equal size.) Under prioritization, the top-ranked subset votes first, and, if it determines a best action, that becomes the decision. If a subset prefers more than one action (a tie), however, then only the tied actions are forwarded to the next subset for consideration, until a decision is made or a tie is broken after the last subset votes. If the user specifies a level of prioritization, additional testing phases with the same testing problems are included in every run. For example, if level 3 is specified, ACE will run five additional testing phases in addition to the ordinary phase: one with 2 subsets for tier 3, one with 3 subsets, one with a ranked list for tier 3, one with the two best tier 3 Advisors, and one with the single best tier 3 Advisor. (Note that prioritization of tier 3 is different from Pusher, because Pusher never uses more than one tier 3 Advisor, whereas if a promoted Advisor fails to make a single choice, the remaining choices will be forwarded to tier 3.)

Prioritization speeds computation because a decision can often be made without devoting cycles to the full complement of Advisors. The number of subsets is open to question, however. In the extreme case, tier 3 effectively becomes a ranked list, and may lose the synergy among good reasons, which accounts for ACE's accuracy. (This is another confirmation that promotion is doomed to failure.) Under prioritization into some (problem class-dependent)

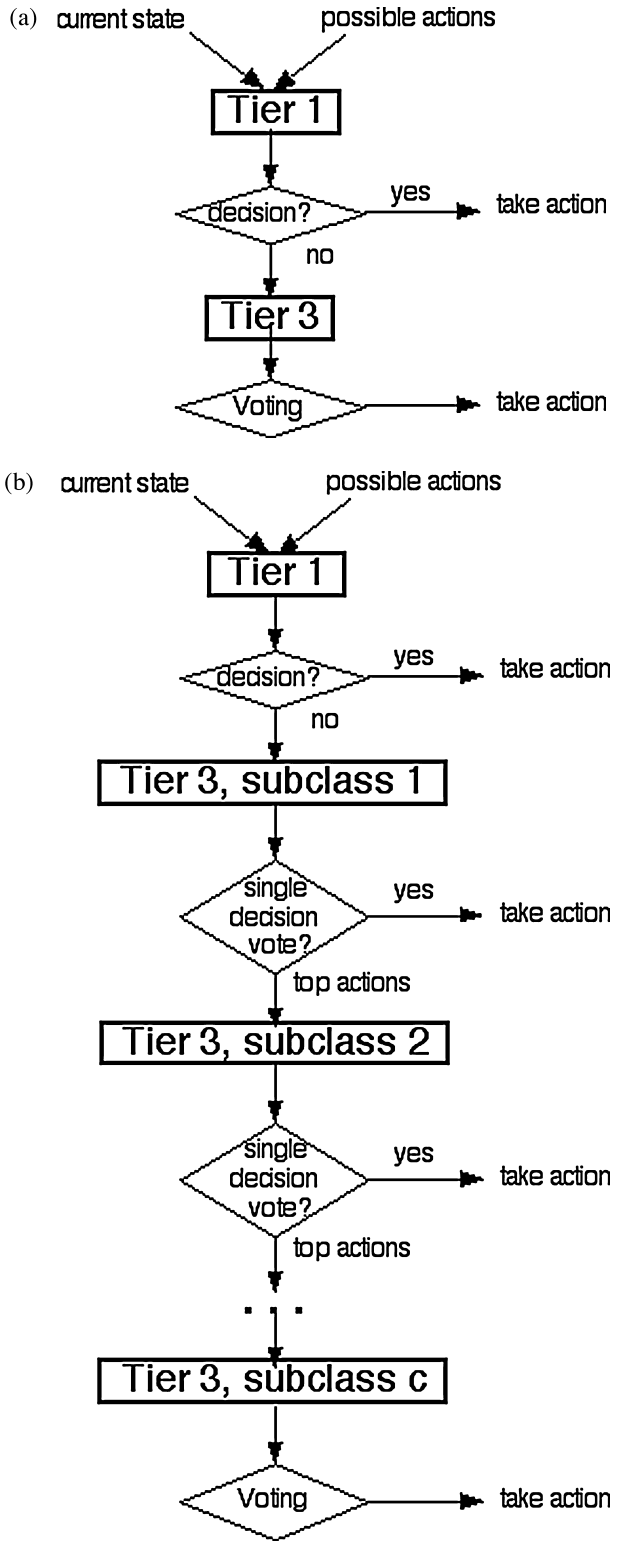


FIGURE 10. Schematics of ACE's decision process (a) without prioritization and (b) with prioritization.

number of subsets, ACE typically reduces its computation time. The most extreme version of prioritization is to consult only the best variable-ordering heuristic and the best value-ordering heuristic or simply the best variable-ordering heuristic with lexical value selection.

These results are readily transferable to solvers other than ACE. Our experience with promotion indicates that a single heuristic, no matter how putatively reliable, will not perform as well as an ACE-like mix. Our results with Pusher indicate that the folklore is indeed correct, and that it is possible to learn the deepest error for a class of problems and use it as a cutoff, after which a single good heuristic (or a random decision) will suffice. Finally, our experience with prioritization demonstrates that ranked lists of heuristics can be less effective than an ACE-like mix.

7. KNOWLEDGE REPRESENTATION

ACE has a large collection of problem class files. A *problem* is represented in ACE as a set of variables, a domain of possible values for each variable, and a set of binary constraints on pairs of variables. A binary constraint on variables x and y is represented as a list of acceptable pairs (a, b), where a is in the domain of x and b is in the domain of y. The representation in the input file for the problem in Figure 1 would number the variables, rather than use letters:

```
(TITLE (GRAPH-1 example))
(PARAMS NUMVARS 4 DMAX 5 EXP-DENSE 0.50 EXP-TIGHT 0.32)
(ARCS ((1 2) (1 4) (3 4)))
(ROOT 4)
(VAR 1 VAR-1 NUM (1 2 3))
(VAR 2 VAR-2 NUM (1 2 4 5 6))
(VAR 3 VAR-3 NUM (1 2))
(VAR 4 VAR-4 NUM (1 3))
(CON (1 2) P (= =) ((1 1) (2 2)))
(CON (1 4) P (= =) ((2 1) (3 1) (3 2)))
(CON (3 4) P (= =) ((1 3) (2 1) (2 3)))
```

During execution, a *state* represents a partial assignment. In the initial state of a problem, no variables have values; in the final state of a successful search, the problem is solved and all the variables have values. A state includes the unassigned variables and their dynamic domains, the assigned variables and their (temporarily) assigned values, the *impossible values* for each unassigned variable (values that have been tried with this partial assignment but found inconsistent with it), and the number of constraint checks incurred during any digressions that generated those impossible values. Each state also stores the edges remaining in the dynamic constraint graph, a pointer to the state immediately prior to the current one during search, and the results from certain time-consuming computations needed by more than one heuristic (e.g., how many variables are now assigned values, the legal actions currently available).

For each problem, ACE retains a *history* of successful decisions, represented as pairs of the form $<s, d>$ where s is a state, d is the decision ACE for that state. ACE also keeps a stack of states in which decisions were made. When a value is retracted, the problem's history is corrected, so that a solved problem's history contains only correct decisions. (For an overconstrained problem, eventually the only retained state is a replica of the initial state but without any remaining alternatives for some variable.) The stack, in addition, retains the roots of digressions destined to become training examples for weight learning.

ACE's useful knowledge records the value and stability of the weights for each Advisor in each stage. It also includes the risk associated with each Advisor and a distribution of the

number of steps required to solve the problems in the class. Finally, the deepest error (see Section 6.6) is recorded as an integer.

7.1. Knowledge about Constraint Solving

A program that learns to solve CSPs should be armed with well-known, empirically demonstrated retraction methods, inference methods, and heuristics. The algorithms noted here make ACE complete, that is, ACE could solve any CSP, given enough computational resources. ACE therefore guarantees feasible, but not necessarily optimal, solutions.

ACE uses *chronological backtracking* for retraction. If there are as-yet-untried values in the domain of the most recently selected variable, it replaces the assigned value with another; otherwise it returns the next most recently selected variable until it identifies one with as-yet-untried values. Chronological backtracking is a simple but popular retraction method for CSP solvers.

The user specifies which of ACE's two inference methods should be used in an experiment to reduce the variables' dynamic domains: forward checking or arc consistency. Both methods are readily understood in the context of the underlying constraint graph, where there is an edge between variables x and y if and only if there is an explicit constraint between them. After a value a is assigned to a variable x, *forward checking* removes from the dynamic domain of every neighbor y of x any value b such that (a b) is unacceptable to the constraint Q4 on x and y. For example, in Figure 1, if variable A is set to 1, then the values 2, 4, 5, and 6 can be removed from the domain of B. Forward checking is relatively fast, as it performs at most $O(k \cdot \text{maximum-degree})$ constraint checks after an assignment. A more costly, but potentially more thorough, method is *arc consistency*, which continues to reduce, in turn, the dynamic domains of the neighbors of variables whose domains have just been reduced by a value assignment. For example, again in Figure 1, if variable D is set to 1, then the value 1 must be removed from the dynamic domain of A, and in turn, 1 will be removed from the dynamic domain of B. Arc consistency may reconsider a variable as its neighbors' domains reduce, so it has complexity $O(ek^3)$, where e is the number of edges in the constraint graph. Neither inference method guarantees that a partial assignment will result in a solution, but arc consistency often eliminates more values from subsequent consideration. Whichever inference method ACE employs, it always performs a single arc consistency pass once, before solving begins.

As described in Section 4, ACE has 21 variable-ordering heuristics (20 in tier 3) and 19 value-ordering heuristics (18 in tier 3), listed in the Appendix. It also has some rudimentary graph theory knowledge (e.g., efficient algorithms to identify connected, acyclic, and tree-structured components), and some constraint-solving knowledge (e.g., an algorithm that computes a backtrack-free, static variable ordering for an arc-consistent tree) (Freuder 1982).

7.2. Languages for Heuristics

Off-the-shelf algorithms (e.g., "minimize the dynamic domain of the variable"), cannot adapt to a problem class—they perform in the same way no matter how many problems they encounter. Because both heuristics and search methods have been shown empirically to vary in their efficacy with the problem class (Borrett, Tsang, and Walsh 1996), very little of this knowledge can be treated as if it were correct. Furthermore, as Section 6.4 demonstrated, additional heuristics and methods await discovery.

A programmer may specify a language in which ACE has to learn new heuristics. An *Advisor language* consists of a set of terminals, a set of nonterminals, and a regular grammar defined on them. An Advisor language also has an *interpreter* that produces comments with

an expression as if it were an Advisor, and a *summary heuristic* procedure to combine the comments thus produced from a specified subset of the language's expressions. Finally, an Advisor language maintains information about the status of each of its expressions, as detailed in Section 5.3.

The simple language that generated the PDFD heuristic in Section 6.4 used the metrics Domain, Degree, Forward Degree, and Backward Degree. Each expression in this language is of the form: $<metric_1, operator, metric_2, attitude, stage>$ where the metrics are distinct, *operator* is either product or quotient (sums and differences proved weaker in early testing and were eliminated), *attitude* is maximize or minimize, and *stage* denotes either the first 20% of the variables to be bound, the middle 60%, or the last 20%. Two other languages have been tested within ACE on a few problem classes. One is a language for value selection, and the other uses a larger set of variable-ordering metrics. Neither has provided as strong a result as that described here. For further details on Advisor learning, the interested reader is referred to Epstein, Gelfand, and Lock (1998).

8. DISCUSSION AND FUTURE WORK

ACE is a long-term collaborative project between constraint-solving experts and a team working to refine FORR, the cognitive architecture that supports ACE. Together we continue to identify issues that spur additional research. This section describes the issues that set our current agenda.

FORR decouples reasoning methods from representation. Although representation can provide powerful insights into the nature of a problem, there is no reason to require all of a system's heuristics to compute from a common representation. The current problem-solving state, we argue, should provide as many representations as the system's heuristics require. (For example, a CSP state can be represented as a labeled graph, as well as sets of variables, constraints, and domains.) When more than one Advisor employs the same representation, FORR computes the representation only once and stores it, so that it is readily available to any other Advisors. In this way, ACE does not credit or blame a representation, only the Advisors that reference it. Furthermore, if no successful Advisor ever employs a particular representation, ACE should eventually note that, drop the representation, and redirect the computational resources once devoted to it. ACE currently computes only metrics that will be used by Advisors.

FORR also decouples learning methods from the reasoning process. An autonomous system should acquire only knowledge that it can subsequently apply. ACE's skill lies not in the agglomeration of information but in the intelligent application of that knowledge once it is acquired. ACE applies knowledge only when an Advisor references it. Thus ACE need not credit or blame a particular learning method, only the Advisors that reference the knowledge it acquires. Indeed, if no successful Advisor ever employs knowledge derived from a particular method, then FORR should eventually note that and drop both the method and the knowledge it produces, again conserving computational resources.

The notion of ACE as a test bed is important. FORR's modularity makes it straightforward to simulate other solvers within the ACE framework, and thereby make comparisons with the same hardware and the same computational overhead. Throughout this paper, data on non-ACE algorithms was produced by testing in that manner. We argue, however, that methods that improve ACE are likely to transfer to other solvers. The work on learning new heuristics is an example of that approach.

For now, ACE is not expected to be the fastest constraint solver available. There are two primary reasons for this. First, ACE gathers extensive statistics on its behavior during and after

problem solving, and these computations are time consuming. This is why all comparisons are done by simulation within ACE. Second, many of the metrics for tier 3 rely on one-ply lookahead. To support them, ACE computes all the potential children of each current state. (This is less wasteful than it may seem at first glance, because it caches a variety of computed values required by the Advisors.) ACE is often more accurate (i.e., makes fewer retractions) than simple algorithms (e.g., Minimize Domain for variable selection and select values at random), but it is slower. The price for reasoning correctly appears to be some slowdown. That motivated the work on Pusher, promotion, and prioritization; the tradeoff between speed and accuracy continues to be a focus in our current work.

8.1. Hard Problems and Learning

A learning program in a space whose problem difficulty has a heavy-tailed distribution can be lulled into a false sense of security. Its experience may suggest that it has learned to solve these problems well, but it may not yet have encountered those whose difficulty lies on the hard tail. As noted earlier, there are enough challenging problems in a CSP class, and their range of difficulty is sufficiently broad, to guarantee that eventually the learner will encounter a problem far more difficult than those it has seen thus far. As a result, we anticipated, and have experienced, two major issues in resilient learning for problems with a heavy-tailed distribution of difficulty: making the training set representative of the problem class, and recovery when a problem is far more challenging than any previously encountered.

The learning algorithm described here is responsive both to the cost of an error and to the overall current system performance. These techniques are applicable beyond the scope of this paper. An independent version of DWL (one that ignores whether a decision was for a variable or a value) is readily constructed. Such an algorithm is applicable in any set of problem classes where an error can be pinpointed and its cost can be identified. Q5

When ACE has difficulty in solving a problem, there are several likely explanations: the problem may really be difficult (i.e., from the hard tail of the distribution); ACE may not yet have learned enough about the problem class; or a tie-breaking decision may have led the program in the wrong direction. We are currently investigating four possible responses to such a difficulty:

- There are many CSP inference and retraction methods, each of which has been experimentally observed to curtail search on some problem classes. Other inference methods (e.g., path consistency) can remove inconsistent values from domains. Other retraction methods (e.g., backjumping, backmarking) can prevent repeated exploration of doomed value combinations (Gaschnig 1979; Ginsberg 1993; Prosser 1993; Kondrak and van Beek 1995). Implementation of several of these methods for ACE is now underway. Each of them, however, incurs additional resource cost with no guarantee of improved performance. Our ultimate intention, therefore, is to have ACE learn which inference method and which retraction method to use, and at what point in the search to use them.
- In early work, we relied on a user-specified step limit to terminate search during learning—if ACE was having difficulty, after it reached the step limit it abandoned the problem and went on to the next problem (Epstein and Freuder 2001). Our thesis was that good solutions would be short, and that those were the solutions from which ACE should learn. For example, for problems with 30 variables, whose perfect solution would require only 60 steps (30 variable selections and 30 value selections), we would set a step limit of 100 (Epstein et al. 2002). Although this served well at the time, there was a clear tradeoff: a lower step limit produced training examples from good solutions, but ACE solved fewer problems and thus had fewer training examples (probably elicited after easier problems)

from which to learn. As a result, we had to set the step limit experimentally, and vary it with the class difficulty. As we continue to strengthen ACE and pare its resource requirements, we now find that we can set an arbitrarily high step limit (currently 10,000) and rarely reach it. This permits us to learn from more difficult problems and to overcome (to some extent) the impact of an unlucky tie-breaker.

- Another device we continue to use, although it was not employed in the experiments described here, is *bootstrap learning* between classes (Epstein et al. 2002). Because most of its tier 3 Advisors are constructed as duals, ACE begins with both good heuristics and *antiheuristics*, those that give generally poor advice in a problem class. Some problems are so difficult that ACE solves very few of them during learning, and therefore has few training examples. Instead, with *bootstrap learning*, we train ACE initially on a similar class that is easier (either because κ is not near 1, or because n is much smaller), and then begin on the more difficult class with the weights learned on the easier one. This is the device we are currently using to learn quasigroups with holes, a special case of scheduling problems that can be notoriously difficult. We plan to investigate the merits of bootstrapping through a sequence of classes, rather than merely two.
- Randomly restarting search (with an arbitrary variable and an arbitrary value for it) is effective on difficult CSPs (Gomes et al. 2000). That research restarted after four retractions on a problem, and found 1 million restarts to be an effective method on large quasigroup problems. The user can specify that ACE restart on a problem with which it is having difficulty, as measured by the number of retractions during search. This setting can be merely a counter (e.g., no more than 100 restarts), or should allow for additional search on each new attempt (e.g., no more than 100 restarts with two more retractions permitted on each successive attempt). Furthermore, the user can specify that ACE restart an entire learning phase if it appears to be going badly (e.g., restart if fewer than 10% of the first 20 problems are solved). In addition to random restart, the ACE user may specify other restart techniques that use ACE's knowledge to select a likely variable, a likely value, or both for the first assignment on each restart. Like stage breaks, restart proves effective on some problem classes (generally the more difficult ones), but requires experimentation to arrive at an appropriate setting. We plan to have ACE learn appropriate restart parameters instead.

8.2. Cognitive Orientation

As used here, "cognitively oriented" means that FORR's reasoning structure emulates approaches readily observable in human problem solving, highly effective approaches not always found in traditional AI artifacts. In tier 3, FORR models human experts who simultaneously entertain a variety of (imperfect) rationales for taking an action (Crowley and Siegler 1993; Schraagen 1993; Biswas et al. 1995; Ratterman and Epstein 1995; Keim et al. 1999).

A cognitively oriented avenue of current research is *fast and frugal reasoning*, a paradigm observed in human problem solving. Under limited time, there exists a tradeoff between the speed of decision making and the correctness of the decision. When pressed for time, people may limit their search for information to guide them in the decision process with *noncompensatory* strategies, those that use a single heuristic to prefer a single option (Gigerenzer and Goldstein 1996). People appear to work from a collection of cognitive mechanisms for inference in specific domains (Gigerenzer, Todd, and Group 1999), which includes low-order perceptual and memory processes, including fast and frugal strategies that may be combined to account for higher-level mental processes. Preliminary experiments indicate that fast and frugal reasoning can save computation time in ACE, but the particular variant employed is

related to problem difficulty (Epstein and Ligorio 2004). We expect to have ACE learn which variant, if any, to employ during testing.

8.3. Related Work

Because ACE does unsupervised learning through trial and error with delayed rewards, it qualifies as a reinforcement learner (Sutton and Barto 1998). Ordinarily, reinforcement learning learns a *policy*, a mapping from estimated values of repeatedly experienced states to actions. In learning to solve a class of constraint problems, however, one is unlikely ever to revisit a state, given the large size of both an individual problem's search space and the size of a problem class. Instead, ACE learns a policy that tells it how to act in any state, one that *combines* action preferences as expressed by its Advisors' comments.

FORR offers a variety of features that made it, rather than other well-known architectures, a good candidate for this work. Because one goal of the ACE project is to export knowledge to other solvers, the architecture must provide explicit information about the efficacy of its heuristic procedures. SOAR (Rosenbloom, Newell, and Laird 1991) and Prodigy (Carbonell, Knoblock, and Minton 1991) do not. ACE is expected to learn from experimentation, which neither SOAR nor Theo (Mitchell et al. 1991) is intended to do. Theo would also be inappropriate because its caching is known to produce very large knowledge bases, which would be intractable for most CSPs.

STAGGER (Schlimmer and Fisher 1986) was a program that learned new expressions from its original terms, and is therefore reminiscent of ACE's ability to learn new heuristics from its original metrics. STAGGER's learning, however, was failure-driven, and produced Boolean classifiers, whereas ACE is success-driven and learns a search control preference function for a sequence of decisions in a class of problems. STAGGER did supervised learning, and represented its results as weighted Booleans, while ACE does unsupervised learning of a weighted linear function. STAGGER also had an initial bias and real valued attributes; ACE's initial bias is under construction, and it currently only accepts discrete values.

Both SAGE.2 (Langley 1985) and ACE learn search control from unsupervised experience, reinforce decisions on a successful path, gradually introduce new factors, specify a threshold, and can transfer their abilities to harder problems. SAGE.2, however learns repeatedly on the same task, reinforcing repeating symbolic rules, while ACE learns on different problems in a specified class, reinforcing the originators of correct comments. When SAGE.2 fails, it revises its rules uniformly; when ACE errs, it reduces its weights in proportion to the size of the error. SAGE.2 learns during search, and compares states, but it lacks ACE's random benchmarks. ACE, in contrast, learns only after search and does not compare states.

Most "learning" in CSP programs is mere memorization of *no goods*, combinations of variable-value bindings for a single problem that produce an inconsistency (Dechter 2003). Solnon (2002) used an artificial ant colony to lay trails of pheromones that learn to guide search . It requires that the number of constraints violated can be evaluated incrementally, and is therefore problem-specific, rather than problem–class-specific. The only other substantial effort to learn to solve constraint problems of which we are aware was MULTITAC (Minton 1996). MULTITAC generated Lisp programs that were specialized solvers constructed from low-level semantic components. A MULTITAC solution was also directed to a class of problems, but MULTITAC required that the user first represent each problem in first-order logic. MULTITAC processed only 10 training instances and used a hill-climbing beam search through the plausible control rules it generated, to produce efficient constraint-checking code and select appropriate data structures. Unlike ACE, its resultant algorithms were tie-breaking

rather than collaborative, and its structure lacked the fluidity and flexibility of FORR. It had, for example, no stages, no voting, and no discounting.

ACE is not, in the immediate future, the answer to all CSP researchers' needs. In particular, it currently handles only unary and binary constraints, it does not do local search (but is, therefore, complete), it does not do constraint optimization or soft constraints, and it has not been applied to SAT problems.

8.4. More Work in Progress

The demonstration that ACE can learn to solve hard problems began in Section 6 with what we call "vanilla ACE," one without the benefits we have demonstrated later in this paper from stages, prioritization, pushing, learning to stability, and learning new Advisors. Each of these has in turn been shown to improve performance. The obvious course, therefore, would be to combine them all, with the expectation that ACE would perform even better. That is not, however, what happens. For example, multiple stages actually impair ACE's performance on some classes of problems, and prioritization does not always improve it. Instead, there appears to be, as one might expect in a problem area this ornery, a level no solver, however clever, surpasses. (Indeed, with NP-hard lurking, this should come as no surprise.) ACE provides a still-growing test bed of reasoning methods for CSPs. We are currently experimenting with a more adaptive framework that, before and after the learning phase, but prior to testing, determines on its own which enhancements are productive for a particular problem class.

In addition to the efforts described earlier, we intend to expand ACE's repertoire of problem classes and Advisor languages. We have under development several new weight-learning algorithms, more sophisticated versions of DWL. (Clearly weight learning is essential to ACE's improved performance, because only with weights is prioritization possible.) Work is now underway to have ACE itself learn the number and location of its stage breaks automatically. The selection of an Advisor to push is clearly problem class-dependent; work is in progress to have ACE learn what to push in addition to when to push. We have begun work to jumpstart learning with initial weight biases estimated by ACE for a given problem class as a preliminary phase to an experiment, before the runs begin. Also, although ACE is meant to support constraint solvers rather than replace them, faster processing will expedite discovery. Thus we continue to seek methods that accelerate ACE, with the expectation that, as PDFD did, they will accelerate other solvers too.

Finally, we have also postulated a set of generally-applicable *meta-heuristics* (knowledge about heuristics) intended for mixture of experts systems such as ACE (Epstein 2004). These include accuracy, stability, utility, influence, novelty, acuity, involvement, and risk. ACE currently measures and monitors all these meta-heuristics for each of its Advisors. As described here, ACE currently uses accuracy, stability, and risk in its weight learning. Work is underway to apply the others in a variety of ways, including learning when to prioritize and into how many classes. Although a FORR-based program is intended to tolerate dependent Advisors, multiple Advisors that compute the same, or nearly the same, measure, can distract the weight learner. As ACE's Advisor list grows, we therefore intend to have FORR apply these meta-heuristics to sets of dependent Advisors and select the best from among them.

9. CONCLUSIONS

Although a wide variety of problems can be cast as constraint problems and solved within ACE, the long-term goal of the ACE project is to understand how cognitive architectures can be used to build advanced programming tools. Many of the devices described here for ACE

are actually provided by FORR, including the decision-making structure, weight learning to stability, stages, bootstrap learning, learning heuristics from a language for them, promotion, prioritization, and the construction and monitoring of experiments.

ACE's current achievements include its ability to learn to solve hard problems, to learn new heuristics, and to transfer knowledge to harder problem classes. ACE also knows when to stop learning, and can learn when it is relatively safe to reason less.

Real-world problems, and those that imitate them, are less responsive to off-the-shelf algorithms, which do not adapt to the problems they confront. ACE's results on geometric and small world problems are intended to be more realistic than merely random ones.

ACE can currently support constraint programmers in a variety of ways. It provides incisive results on popular heuristics applied to particular problem classes, and it can characterize a problem class by the heuristics to which it is (and is not) responsive. These include the proportion of time devoted to inference, to retraction, and to decision making, as well as the depth in the search tree at which errors arose, and how severe those errors were. ACE publishes its discoveries of effective combinations of heuristics. Its results are readily exportable, and its meta-knowledge about heuristics should be particularly useful. It also permits researchers to test new heuristics and new combinations of heuristics on a broad variety of problem classes. ACE is envisioned as a research partner, and is proving to be an effective one.

ACKNOWLEDGMENTS

This work was supported in part by NSF IIS-0328743, by PSC-CUNY, by Enterprise Ireland under grant no. SC/2002/0137, and is based upon works supported in part by Science Foundation Ireland under grant 00/PI.1/C075. Thanks to Mark Hennessey, Tiziana Ligorio, Anton Morozov, Menachem Pastreich, Smiljana Petrovic, Stoycho Stoev, Matt Szenher, CUNY's FORR study group, and the Cork Constraint Computation Centre for their support in this work.

REFERENCES

BECK, J. C., P. PROSSER, and E. SELENSKY. 2003. Vehicle routing and job shop scheduling: What's the difference? *In* Thirteenth International Conference on Automated Planning and Scheduling –ICAPS03. **Q6**

BESSIÈRE, C., and J.-C. RÉGIN. 1996. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. *In* Principles and Practice of Constraint Programming - CP96, LNCS 1118. *Edited by* E. C. Freuder. Springer-Verlag, Berlin, pp. 61–75.

BISWAS, G., S. GOLDMAN, D. FISHER, B. BHUVA, and G. GLEWWE. 1995. Assessing design activity in complex CMOS circuit design. *In* Cognitively Diagnostic Assessment. *Edited by* P. Nichols, S. Chipman. and R. Brennan. Erlbaum, Hillsdale, NJ, pp. 167–188.

BORRETT, J., E. P. K. TSANG, and N. R. WALSH. 1996. Adaptive constraint satisfaction: The quickest first principle. *In* 12th European Conference on AI, Budapest, Hungary.

CARBONELL, J. G., C. A. KNOBLOCK, and S. N. MINTON. 1991. PRODIGY: An integrated architecture for planning and learning. *In* Architectures for Intelligence, *Edited by* K. VanLehn. Erlbaum, Hillsdale, NJ.

CHATTERJEE, S., and S. CHATTERJEE. 1987. On combining expert opinions. American Journal of Mathematical and Management Sciences, **7**(3–4):271–295.

CHEESEMAN, P., B. KANEFSKY, and W. M. TAYLOR. 1991. Where the REALLY hard problems are. *In* Twelfth International Joint Conference on Artificial Intelligence, Sydney, Australia.

CROWLEY, K., and R. S. SIEGLER. 1993. Flexible strategy use in young children's tic-tac-toe. Cognitive Science, **17**(4):531–561.

DECHTER, R. 2003. Constraint Processing. Morgan Kaufmann, San Francisco, CA.

EPSTEIN, S. L. 1995. On heuristic reasoning, reactivity, and search. *In* Fourteenth International Joint Conference on Artificial Intelligence, Morgan Kaufmann, Montreal.

EPSTEIN, S. L. 1998. Pragmatic navigation: Reactivity, heuristics, and search. Artificial Intelligence, **100**(1–2):275–322.

EPSTEIN, S. L. 2001. Learning to play expertly: A tutorial on Hoyle. *In* Machines That Learn to Play Games. *Edited by* J. Fürnkranz and M. Kubat, Nova Science, Huntington, NY, pp. 153–178.

EPSTEIN, S. L. 2004. Metaknowledge for autonomous systems. AAAI Spring Symposium on Knowledge Representation and Ontology for Autonomous Systems, AAAI. **Q7**

EPSTEIN, S. L., E. C. FREUDER, R. WALLACE, A. MOROZOV, and B. SAMUELS. 2002. The adaptive constraint engine. *In* Principles and Practice of Constraint Programming — CP2002. *Edited by* P. Van Hentenryck, Springer Verlag, Berlin, LNCS **2470**, pp. 525-540.

EPSTEIN, S. L., and G. FREUDER. 2001. Collaborative learning for constraint solving. *In* Principles and Practice of Constraint Programming — CP2001. *Edited by* T. Walsh. Springer Verlag, Berlin, **2239**, pp. 46–60.

EPSTEIN, S. L., J. GELFAND, and E. T. LOCK. 1998. Learning game-specific spatially-oriented heuristics. Constraints, **3**(2–3):239–253.

EPSTEIN, S. L., and T. LIGORIO. 2004. Fast and frugal reasoning enhances a solver for really hard problems. Cognitive Science 2004, Earlbaum, Chicago.

FREUDER, E. 1982. A sufficient condition for backtrack-free search. Journal of the ACM, **29**(1):24–32.

FREUDER, E., and A. MACKWORTH, Eds. 1992. Constraint-Based Reasoning. MIT Press, Cambridge, MA.

FROST, and R. DECHTER. 1995. Look-ahead value ordering for constraint satisfaction problems. *In* IJCAI-95, Montreal.

GASCHNIG, J. 1979. A problem similarity approach to devising heuristics: First results. *In* Sixth International Joint Conference on Artificial Intelligence, Morgan Kaufmann, Tokyo.

GEELEN, P. A. 1992. Dual viewpoint heuristics for binary constraint satisfaction problems. *In* 10th European Conference on Artificial Intelligence. **Q8**

GENT, I. E., E. MACINTYRE, P. PROSSER, and T. WALSH. 1996. The constrainedness of search. In Thirteenth National Conference on Artificial Intelligence. **Q9**

GIGERENZER, G., and G. GOLDSTEIN. 1996. Reasoning the fast and frugal way: Models of bounded rationality. Psychological Review, **103**(4):650–669.

GIGERENZER, G., P. M. TODD, and A. R. GROUP. 1999. Simple Heuristics that Make Us Smart. Oxford University Press, New York.

GINSBERG, M. L. 1993. Dynamic backtracking. JAIR, **1**:25–26.

GOMES, C. P., B. SELMAN, N. CRATO, and H. KAUTZ. 2000. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. Journal of Automated Reasoning, **24**:67–100.

JACOBS, R. A. 1995. Methods for combining experts' probability assessments. Neural Computation, **7**: 867–888.

JOHNSON, D. B., C. R. ARAGON, L. A. MCGEOOH, and C. SCHEVON 1989. Optimization by simulated annealing: An experimental evaluation: Part 1, Graph partitioning. Operations Research, **37**:865–892. **Q10**

KEIM, G. A., N. M. SHAZEER, M. L. LITTMAN, S. AGARWAL, C. M. CHEVES, J. FITZGERALD, J. GROSLAND, F. JIANG, S. POLLARD, and K. WEINMEISTER. 1999. PROVERB: The probabilistic cruciverbalist. *In* Sixteenth National Conference on Artificial Intelligence. AAAI Press, Orlando.

KIZILTAN, Z., P. FLENER, and B. HNICH. 2001. Towards inferring labelling heuristics for CSP application domains. *In* KI'01. Springer-Verlag, Berlin.

KONDRAK, G., and P. VAN BEEK. 1995. A theoretical evaluation of selected backtracking algorithms. *In* Fourteenth International Joint Conference on Artificial Intelligence. **Q11**

LANGLEY, P. 1985. Learning to search: From weak methods to domain-specific heuristics. Cognitive Science, **9**:217–260.

MACKWORTH, A. K. 1977. Consistency in networks of relations. Artificial Intelligence, **8**:99–118.

MINTON, S. 1996. Automatically configuring constraint satisfaction programs: A case study. Constraints, **1**(1/2):7–43.

MITCHELL, T., J. ALLEN, P. CHALASANI, J. CHENG, O. ETZIONI, M. N. RINGUETTE, and J. C. SCHLIMMER, Eds. 1991. Theo: A framework for self-improving systems. *In* Architectures for Intelligence. Erlbaum, Hillsdale, NJ.

NUDEL, B. 1983. Consistent-labeling problems and their algorithms: Expected-complexities and theory-based heuristics. Artificial Intelligence, **21**:135–178.

PROSSER, P. 1993. Hybrid algorithms for constraint satisfaction problems. Computational Intelligence, **9**(3):268–299.

RATTERMAN, M. J., and S. L. EPSTEIN. 1995. Skilled like a person: A comparison of human and computer game playing. *In* Seventeenth Annual Conference of the Cognitive Science Society, Erlbaum, Hillsdale, NJ.

ROSENBLOOM, P. S., A. NEWELL, and J. E. LAIRD. 1991. Toward the knowledge level in soar: The role of the architecture in the use of knowledge. *In* Architectures for Intelligence. *Edited by* K. VanLehn. Erlbaum, Hillsdale, NJ.

RUAN, Y., E. HORVITZ, and H. KAUTZ. 2004. The backdoor key: A path to understanding problem hardness. *In* AAAI-2004, AAAI Press, San Jose, CA.

SABIN, D., and E. C. FREUDER. 1997. Understanding and improving the MAC algorithm. *In* Principles and Practice of Constraint Programming. Springer Verlag, Berlin, pp. 167–181.

SCHLIMMER, J. G., and D. FISHER. 1986. A case study of incremental concept induction. *In* Fifth National Conference on Artificial Intelligence, Philadelphia.

SCHRAAGEN, J. M. 1993. How experts solve a novel problem in experimental design. Cognitive Science, **17**(2):285–309.

SOLNON, C. 2002. Boosting ACO with a preprocessing step. *In* EvoWorkshops 2002: Applications of Evolutionary Computing. **Q12**

SUTTON, R. S., and A. G. BARTO. 1998. Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA.

TSANG, E. P. K. 1993. Foundations of Constraint Satisfaction. Academic Press, London.

WALSH, T. 1999. Search in a small world. *In* IJCAI-99, Morgan Kaufmann, Stockholm.

APPENDIX

Other than those learned during problem solving, ACE begins with a set of Advisors selected from the following. "Any" here denotes either random or lexical selection, as determined by the user, and is applied uniformly for all Advisors. The following list is restricted to Advisors used to generate results cited in this paper. The list is continually evolving, however.

Tier 1 (in Order)

Victory. When only a single variable has no assigned value and has been selected, Victory comments in favor of any value in the dynamic domain of that variable.

Degree Zero. When a variable is to be selected next, Degree Zero vetoes any variable whose dynamic degree is zero. This Advisor is not used if Later is.

Later. When a variable is to be selected next, Later vetoes any variable whose forward degree is less than its dynamic domain. This Advisor is intended only for graph coloring, and is not used if Degree Zero is.

Unique Value. When a variable is to be selected next, Unique Value forces the selection of any variable whose dynamic domain contains exactly one value.

Pusher. When a variable is to be selected next, Pusher uses only the single best tier 3 Advisor to make a decision. If the Advisor does not comment, or if no variable-selection Advisor survives into testing, Pusher selects any variable. See Section 6.6 for further details.

Tier 3 (Variable Selection)

Min/Max Static Degree. Supports variables in increasing/decreasing order of their static (in the original constraint graph) degree.

Min/Max Domain. Supports variables in increasing/decreasing order of their dynamic domain size. Smaller domains are likely to fail sooner.

Min/Max Domain/Degree. Supports variables in increasing/decreasing order of the ratio of their dynamic domain size to their static degree. (Bessière and Régin 1996)

Min/Max Backward Degree. Supports variables in increasing/decreasing order of the number of their valued neighbors.

Min/Max Forward Degree. Supports variables in increasing/decreasing order of their dynamic degree (number of unvalued neighbors).

Min/Max Value Pairs. Supports variables in increasing/decreasing order of the number of pairs of values with their neighbors still supported by the current partial assignment (Kiziltan, Flener, and Hnich 2001).

Min/Max Static Connected Edges. Orders the edges in the original constraint graph descendingly, by the sum of the degrees of their vertices. Supports variables in increasing/decreasing order of their incidence on these edges, with preference for the higher-degree vertex.

Min/Max Static Less Connected Edges. Orders the edges in the original constraint graph ascendingly, by the sum of the degrees of their vertices. Supports variables in increasing/decreasing order of their incidence on these edges, with preference for the higher-degree vertex.

Min/Max Dynamic Connected Edges. Orders the edges in the dynamic constraint graph descendingly, by the sum of the degrees of their vertices. Supports variables in increasing/decreasing order of their incidence on these edges, with preference for the higher-degree vertex.

Min/Max Dynamic Less Connected Edges. Orders the edges in the dynamic constraint graph ascendingly, by the sum of the degrees of their vertices. Supports variables in increasing/decreasing order of their incidence on these edges, with preference for the higher-degree vertex.

Tier 3 (Value Selection)

These Advisors address the assignment of a value to a variable that has already been selected.

Min/Max Common Value. Supports values that have been assigned less/more often in the partial assignment.

Min/Max Options Value. Supports values associated with the least/most values for the neighbors of the variable.

Min/Max Static Conflicts Value. Minimizes/maximizes based on the number of values that would be supported in the static domains of the unvalued neighbors of the variable (Frost and Dechter 1995).

The following two pairs of Advisors address the domain size of the future variables after application of an inference method.

Min/Max Small Domain Value. Minimizes/maximizes the minimum domain size (Frost and Dechter 1995).

Min/Max Product Domain Value. Minimizes/maximizes the product of the domain sizes (Geelen 1992).

The following two pairs of Advisors base their comments on the values supported in the dynamic domains of the neighbors of the variable after application of an inference method. Recall that an inference method considers ordered pairs of values for edge labels; those are distinguished here from the individual values that appear in those pairs.

Min/Max Secondary Pairs Value. Minimizes/maximizes the number of pairs of values.

Min/Max Secondary Value. Minimizes/maximizes the number of distinct values.

The following two pairs of Advisors base their comments on the minimum domain size of the future variables after the application of an inference method (Frost and Dechter 1995).

Min/Max Weighted Domain Score Value. Minimizes/maximizes domain size, weighted by the number of future variables with domains of that size.

Min/Max Domain Score Value. Minimizes/maximizes the largest domain, treating the number of future variables with domains of that size as an exponent.

QUERIES

Q1 Au: Please check that the change retains the intended sense.

Q2 Au: Please check this usage—‘ . . . consulted and the comment.

Q3 Au: Please confirm whether there is an apostrophe or prime after ACE.

Q4 Au: Please check this usage—‘every neighbor *y* of *x* any value *b*’.

Q5 Au: The sentence—‘ These techniques...’ is not clear. Please check.

Q6 Au: Please provide location of the conference.

Q7 Au: Please provide location for the symposium.

Q8 Au: Please provide location of the conference.

Q9 Au: Please provide location of the conference.

Q10 Au: Please confirm the page range in this reference.

Q11 Au: Please provide location of the conference.

Q12 Au: Please provide location of the conference.