A Cognitively-Oriented Architecture Confronts Hard Problems

Susan L. Epstein
Hunter College and The Graduate School of The City University of New York
susan.epstein@hunter.cuny.edu

Abstract

This paper describes a cognitively-oriented architecture that facilitates the development of expertise. Based on knowledge about human decision making, it integrates multiple representations, multiple decision-making rationales, and multiple learning methods to support the construction of intelligent systems. A constraint solver implemented within the architecture engineers a problem-solving paradigm. This program manages a variety of search heuristics and learns new ones. It can transfer what it learns on simple problems to solve more difficult ones, and can readily export its knowledge to ordinary solvers. It is intended both as a learner and as a test bed for the constraint community. Both the program and the architecture are ambitious, ongoing research projects to support human reasoning.

The thesis of this work is that AI architectures for learning and problem solving can benefit from what works for people. Since people function remarkably well in a complex world, other problem solvers may profit both from descriptions of human problem solving and from the devices that people rely upon to achieve their goals. This paper reports on a cognitively-oriented program that learns to solve difficult constraint problems, has rediscovered an important result in graph coloring, and has learned new heuristics that readily export to improve ordinary CSP solvers (Epstein et al., 2002; Epstein & Freuder, 2001). The program can determine when it has finished learning, and can learn when and how to modify its own reasoning structure without sacrificing performance.

Here, an *architecture* is a system shell within which an application program for a particular *domain* (set of related problem classes) can be constructed. To produce a program, one applies an architecture to a domain by providing it with knowledge: what to learn, how and when to learn it, and how to reason in the domain. An architecture is *cognitively-oriented* if it simulates significant characteristics of human problem solving.

Since a human expert solves certain problems faster and better than other people (D'Andrade, 1990), programs produced by a cognitively-oriented architecture that facilitates the development of expertise should eventually solve faster and better than others too. Traditionally, an architecture engineers a domain-specific program. The application described here, however, formulates knowledge about constraint satisfaction. Thus it *engineers a problem-solving paradigm*, rather than a single domain. Furthermore, one can specialize the program further, say, for graph coloring or scheduling.

The next section references some cognitive science results and interprets them for an architecture. For clarity, we initially illustrate these ideas with an autonomous mobile robot moving toward a goal object in a dynamic environment. Subsequent sections describe how those results influence the formulation of a particular cognitively-oriented architecture, and how that architecture supports a program that learns to solve constraint problems. The final section discusses related and current work.

Some Contributions from Cognitive Science

The results discussed below are particularly relevant in the creation of a cognitively-oriented architecture, and in any attempt to understand or communicate with one.

Knowledge

Human knowledge is represented in a variety of formats. People retain sensory memories (e.g., the smell of a hospital), procedural data (e.g., how to ride a bicycle), and declarative information (e.g., correct spelling). People also rely on their senses to focus their attention and to cue their decisions. When masters at the game of Go look at a board, for example, eye-tracking data indicate that they very quickly identify a few crucial possible moves (Yoshikawa, Kojima & Saito, 1998). This ability to attend to visual features, to determine their significance, and to value them accordingly is called *spatial cognition*. It is why people play chess better *without* blindfolds — vision not only focuses their attention, but is also an integral part of decision making.

Much of human knowledge is merely what we call *useful knowledge* here; it is only probably correct and possibly applicable to future decisions (e.g., a particular opening for a board game). Nonetheless, people comfortably employ useful knowledge to make decisions, even when some of it is inconsistent. For example, a high likelihood of rain and a dislike of getting wet do not always convince one to carry an umbrella. Furthermore, brain-lesion studies show that multiple paths to much of our knowledge are formed according to several factors (e.g., sound, color, frame of reference) (Sacks, 1985). As a result, when one access route is blocked, another may still be viable.

Decision making

To *satisfice* is to make decisions that are good enough in a simple model of a complex world (Simon, 1981). For example, a resident of Manhattan about to return home from work does not typically contemplate the likelihood of a flood or a blackout — she merely walks to the subway. Because blackouts and floods are unlikely in her environment, it does not pay to model them. When choosing among algorithms for a program, the most accurate one may be intractable (e.g., minimax for chess), or merely of low *utility* (roughly, return on expended effort). A program is said to be *limitedly rational* whenever it replaces a more accurate reasoning method with a heuristic one. Such a program satisfices.

Most interesting problems require not a single decision but a sequence of them. Even a reasoning method judged valuable on a particular problem may not be equally good throughout that sequence. Chess masters, for example, partition a game into three *stages*: the opening (roughly the first 20% of the moves), the endgame (roughly the last 20% of the moves), and the middlegame (the remainder). Chess players make decisions differently in these stages, almost as if they were different games.

Moreover, one rationale for decision making is rarely enough. People appear to work from a collection of cognitive mechanisms for inference in specific domains, including low-order perceptual and memory processes (Nichelli et al., 1994). Human experts simultaneously entertain a variety of (imperfect) domain-specific rationales for taking an action, and introduce new ones gradually (Biswas et al., 1995; Crowley & Siegler, 1993; Keim et al., 1999; Ratterman & Epstein, 1995; Schraagen, 1993). These rationales vary in their accuracy, as, for example, when a chess player wants to control the center of the board, and also wants to trap the king. As nearly as psychologists can detect, these rationales are *not* ranked lists of general rules, but a conglomeration of domain-specific principles, some more reliable than others. Among them, these multiple principles provide synergy in decision making.

Learning

Human expertise is not innate; it develops with practice (Ericsson & Staszewski, 1989). People acquire knowledge in many different ways, and may learn the same item with several different methods. Indeed, recent research suggests that multiple learning methods for the same concepts actually enhance human understanding (Anonymous, under review). Human learning also requires participation, not merely observation. For example, one must play a violin to learn to play it — watching a violinist may be helpful, but it is not enough.

People also create new reasoning methods, which are often flawed (Siegler & Crowley, 1994). Children learning multiple-place subtraction, for example, often invent explanations not presented during instruction, and gradually introduce them into computation. Most of these invented methods are wrong, but their inaccuracy is not revealed until after they have been in use for some time.

Meta-knowledge

People gauge their own performance by a variety of standards, and use those evaluations to modify their behavior. These standards often conflict as, for example, accuracy versus speed. One way to speed computation is to recognize similar situations and treat them uniformly. Thus, people may cache and reuse results of prior computation.

Another way people speed computation is to formulate a set of (possibly ordered or partially-ordered) decisions for some part of the problem (a *subgoal*), with the understanding that, if things do not go well, the entire set can be discarded. In *situation-based reasoning*, people under time pressure use domain knowledge both to identify a context responsive to the selection of a set of decisions, and to dictate the method by which those decisions should be selected (Klein & Calderwood, 1991). That method is not guaranteed to return a correct answer or, indeed, any answer at all. Situation-based reasoning can be thought of as a uniform procedure that triggers in the presence of some context, and responds to it with a set of decisions. Encountering an obstacle directly between our robot and its goal is a situation, which could trigger a procedure to calculate a way to go around the obstacle. The robot's procedure or the execution of its output could be made contingent on adequate progress. Responses like these, formulated to address a particular kind of situation, have proved effective: inexpensive to produce and easy to abandon (Agre & Chapman, 1990; Epstein, 1998).

Another active area of cognitive science research is *fast and frugal reasoning*, a limitedly-rational paradigm observed in human problem solving (Gigerenzer & Todd, 1999). Under certain circumstances, people may limit their search for information to guide them in the decision process with *non-compensatory* strategies, ones that use a single rationale to prefer a single option. Fast and frugal reasoning methods rely on *recognition*, the favoring of familiar objects over unfamiliar ones. Situation-based reasoning may be paraphrased as "I have been in several similar situations before, and I will now mentally test those solutions in turn, until one of them seems as if will work here as well." In contrast, the recognition underlying fast and frugal reasoning may be paraphrased as "I have seen this choice before and will therefore select it, whether or not my current situation is similar to the one in which I made this choice." Fast and frugal methods themselves select only among recognized choices, and then apply a second standard, such as "try the last heuristic I used with this method." Although at first glance fast and frugal reasoning seems more limited than rational, it works surprisingly well on challenging problems (Epstein & Ligorio, 2004). Fast and frugal reasoning enhances, rather than contradicts, the findings on multiple rationales cited earlier. The overall structure is, as we shall see, simply a bit more complicated.

Implications for cognitively-oriented architectures

In summary, a cognitively-oriented architecture should support reasonable behavior. Its programs should solve easy problems quickly; hard problems should take longer. The programs should not make obvious errors, and they should balance accuracy against speed. The architecture itself should:

• Support multiplicity: multiple representations for data, multiple rationales for decision making, multiple learning methods, and multiple stages.
• Support the production and interaction of both individual decisions and sets of them.
• Be robust to error, discarding decisions, sets of decisions, and the rationales that support them.
• Tolerate and reason with inconsistent and incomplete information.
• Decouple data, learning methods, and decision methods from one another.
• Restructure decision making in response to meta-heuristics and to what it learns about a problem class.

Such an architecture must be modular and support the learning of data, of rationales, and of its own structure. As a result, its programs run the risk of error (which may make it unacceptable in certain domains), and must be able to learn in the noisy environment of their own mistakes. The architecture in the next section addresses all these criteria. Its cornerstones are satisficing, useful knowledge, and visual cognition.

FORR

FORR (FOr the Right Reasons) is a cognitively-oriented architecture for learning and problem solving. It actively encourages the use of multiple learning methods, multiple representations, and multiple decision rationales. FORR learns to combine rationales to improve problem solving. It also acquires useful knowledge for each problem class it encounters. It supports situation-based reasoning, and fast and frugal reasoning. It can learn new decision-making rationales on its own, and readily incorporates them into its reasoning structure. In short, it meets the criteria of the previous section.

One applies FORR to a domain by programming the decision-making rationales and the pertinent useful knowledge (what to learn, and when and how to learn it). The best-known FORR-based programs are *Hoyle* (Epstein, 2001), a program that learns to play board games; *Ariadne* (Epstein, 1998), a program that learns to find its way through mazes; and *ACE*, a program that learns to solve constraint satisfaction problems.

Advisors and the decision hierarchy

The building blocks of a FORR-based system are its Advisors. An *Advisor* is a domain-wide but problem-class-independent, decision-making rationale. (For example, Advisors for our robot should theoretically be applicable to paths through warehouses, as well as office buildings.) The role of an Advisor is to support or oppose current legal actions. 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' output. A FORR-based program makes decisions based upon its Advisors' output; FORR controls and measures the learning process.

Both the formulation of Advisors and their organization rely heavily on domain knowledge. Each Advisor is represented as a time-limited (and therefore limitedly-rational) procedure. Uniformity is imposed only on the input and output of these procedures. The input is all useful knowledge learned for the problem class, the current problem state, and some set of legal actions in that state. The output of each Advisor is its *comments*, triples of the form <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). Other than input and output, Advisors may rely on their own specialized knowledge representations and computations.

To *consult* an Advisor is to solicit comments from it, that is, to execute it. For example, assume that our robot can move only to any location currently visible to it, but that obstacles may partially obstruct its vision. An Advisor whose rationale is "make no rash decisions" might comment in favor of nearby locations and oppose distant ones, while an Advisor for "get close" might comment in favor of locations closer to the goal and oppose those farther from it.

Figure 1 summarizes FORR-based decision making. The input is the current state and the legal actions there. If there are no such actions, search terminates; if there is exactly one action, it is executed. Otherwise, decision making moves through a hierarchy, three *tiers* of Advisors categorized by their trustworthiness and whether they focus on individual decisions. Tier-1 Advisors and tier-3 Advisors comment on individual decisions; tier-2 Advisors comment on *subgoal approaches*, sets of actions. The programmer initially assigns each Advisor to a tier; those in earlier tiers take precedence over those in later tiers. Advisors are not required to be independent. Useful knowledge is learned between tasks, accessible to all Advisors, and may be acquired by any 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 solve easy problems 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, an Advisor called *Victory* for our robot would have absolute authority; when the goal is in sight, it would comment to move directly to it. If a tier-1 Advisor has *veto power*, the actions it opposes are eliminated from further consideration by subsequent Advisors. For example, a tier-1 Advisor for the robot might reference useful knowledge about dead-ends, and comment to avoid those known not to include the goal. Tier-1 Advisors are consulted in order of relative importance, as pre-specified by the programmer. A domain-specific version of Victory is always first in the ordering for tier 1.

Tier-1 Advisors are consulted in sequence, until either a

```
Until the problem is solved or proved impossible
   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 and Comments recommend actions R
         then select any action in R and return it
      If A can veto and Comments veto actions V and 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
            If approaches are generated
               then select one and return it
               else continue
Tier 3: For each Advisor A in the unordered tier 3
         collect comments of A on Candidates into Comments
      Return voting(Comments)
   Execute the action
```

Figure 1: FORR's decision-making algorithm.

single decision is selected by an Advisor with absolute authority, or until vetoes reduce the set of candidate actions to a single one. In either event, the selected action is executed to produce the next state. A tier-1 Advisor called *Enforcer* monitors any currently-selected subgoal approach. Enforcer vetoes actions that conflict with the current approach, and discards the approach itself if none of its actions remain. If tier 1 does not make a decision and there is no current subgoal approach, all remaining (not vetoed) actions are forwarded to the Advisors in tier 2, which attempt to generate a subgoal approach.

Tier-2 Advisors and subgoals

During the solution of a problem, unpredictable subgoals may arise. For example, when our robot detects an obstacle dynamically, going around that obstacle could become a subgoal. *Tier-2 Advisors* are rationales that produce a set of actions directed to a subgoal. We call the set of actions a subgoal approach, and the use of tier 2 to address it we call *DSO*, for dynamic subgoal ordering. DSO includes methods that encourage the production and identification of subgoals, methods that order subgoals relative to each other, and methods that formulate subgoal approaches.

The creation of subgoals, selection among them, and the formulation and selection of approaches to them are all domain-dependent. In Ariadne, all this was pre-specified by the programmer; in Hoyle, much of it can be learned (Lock & Epstein, 2004). If tier 2 generates a subgoal approach, control returns to tier 1, where Enforcer will monitor the approach's execution. If neither tier 1 not tier 2 makes a decision, the remaining actions are forwarded to tier 3, where all the Advisors comment in parallel.

Tier-3 Advisors and voting

Tier-3 Advisors are procedures that comment on single actions, without absolute authority, veto power, or any guarantee of correctness. To select an action in tier 3, a FORR-based program combines the comments of its tier-3 Advisors. Each Advisor comments on any number of actions. Unlike tier 1, where Advisors are consulted in sequence, in tier 3 all Advisors are consulted at once, and a weighted combination of their comments (described below) produces a decision. An Advisor whose comments support moving our robot toward the goal is an example of a tier-3 Advisor, because proximity does not guarantee access.

FORR includes several modified, hill-climbing, weight-learning algorithms that reward and penalize decisions after a task is successfully completed (Epstein et al., 2002). A *discount factor* serves to introduce the 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. If no single action is deemed best, one from among the best is chosen with a method specified by the user (i.e., lexical or random tie-breaking). Each tier-3 Advisor has a learned weight and a discount factor.

A tier-3 decision is made by *voting*, a process that combines Advisors' comments to determine the action with the greatest support. (See Figure 2.) Voting multiplies the strength of the opinion of each Advisor on each action by

$$\underset{j}{argmax}\left\{\sum_i q_i\, w_i\, c_{ij}\right\}$$

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

Figure 2: The voting computation in tier 3.

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.

To participate in voting, an Advisor must be consulted and then comment. A *benchmark* is a non-voting, baseline procedure which is presented with the same actions as the Advisors it gauges, and models how well random comments would do on the same decisions. A benchmark comments with randomly-generated strength on *n* randomly-chosen actions $(0.5)^n$% of the time. If our robot represented decisions as location coordinates, there would be a single benchmark; if decisions were either orientation or direction, there would be two benchmarks, one for each. To apply FORR, a programmer specifies useful knowledge items, how and when to learn them, and a set of Advisors. We turn now to an important problem-solving paradigm.

Constraint Satisfaction

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 (Freuder & Mackworth, 1992). 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 (Wallace, 1996), but arriving at appropriate methods in practice is not a trivial cookbook exercise (Beck, Prosser & Selensky, 2003). At present, for each new, large-scale CSP, a constraint programmer must seek the right method combination.

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 (Beck, Prosser & Selensky, 2003; Tsang, 1993). An example of a CSP and its underlying constraint graph appear in Figure 3. (For simplicity, we restrict discussion to binary CSPs, where each constraint involves no more than two variables.) A *solution* for a CSP is a value assignment for all the variables that satisfies all the constraints. Every CSP has an underlying *constraint graph*, which represents each variable by a vertex whose possible labels are its domain values. An edge in the constraint graph appears between two vertices whenever there is a constraint on the variables corresponding to them. 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. As decisions are made, this graph can be viewed dynamically, as in Figure 4.

Four parameters characterize a CSP: <*n, k, d, t*>. Here, *n* is the number of variables in the CSP, and *k* its maximum domain size. The *density d* of a CSP is the fraction of possible edges it includes: $2e/n(n-1)$ for *e* edges. The *tightness t* of a CSP is the percentage of possible value pairs it *ex*cludes from the domains of the endpoints of its edges. The set of all problems with the same values for *n, k, d,* and *t* form a *class* of CSPs. Thus the CSP in Figure 3 is in the class <4, 5, .5, .328>. There are also specializations of CSPs that describe particularly interesting problem classes, such as graph coloring.

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; one with high density and tightness is likely to be *overconstrained* and have no solutions. Although CSP solution 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 & Taylor, 1991), known as the *phase transition*. The best-known estimate of difficulty for a CSP class is *kappa* (Gent et al., 1996). Solvable problem classes with kappa near 1 are said to be at the *complexity peak*, that is, they generally have a single solution that is particularly difficult to find.

ACE

ACE, a project in collaboration with the Cork Constraint Computation Centre, is a FORR-based program for constraint solving. It has a large library of problem classes, each represented by many randomly-generated examples. From generic components, ACE learns to synthesize an effective algorithm adapted to a specific CSP problem class. ACE learns to solve difficult CSPs efficiently. ACE also characterizes different classes of CSPs differently. As a result, ACE's learning can provide guidance in problem classes where ordinary CSP approaches stumble.

In ACE, the solution to a CSP is a sequence of decisions in which one alternately selects a variable and then assigns a value to that selected variable. An Advisor is applicable either to variable selection or to value selection (so there

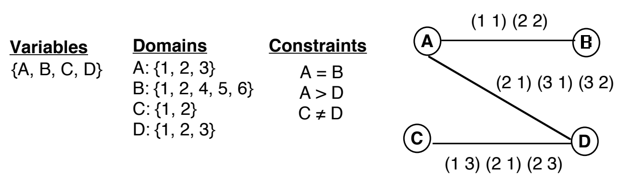

Figure 3: A simple constraint problem and its underlying constraint graph. Labels on the edges give acceptable values for the variables in alphabetical order. For example, (3 1) on AD means A can be 3 when D is 1. This problem has exactly one solution.

are two benchmarks for weight learning). In tier 1, ACE has only a few Advisors. Its version of Victory arbitrarily assigns a remaining, consistent value to the last unbound variable. Another Advisor vetoes the selection of any variable without neighbors in the dynamic constraint graph. In tier 2, ACE's Advisors capitalize on spatial cognition as a person might, seeking to disconnect the graph into nontrivial connected components as in Figure 4. A subgoal approach either disconnects the graph into components or restricts computation to one component at a time.

In tier 3, ACE has an ever-expanding list of Advisors based on CSP lore and on the evolving CSP literature. Some tier-3 Advisors are duals, one of which maximizes a metric while the other minimizes it. A *static* metric is computed once, before problem solving begins; a *dynamic* metric is recomputed periodically during decision making. For example, one traditional static metric for variable selection is the *degree* of a variable, the number of neighbors it has in the original constraint graph. The Advisor *Maximize Degree* supports the selection of unvalued variables in decreasing degree order. 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 solution; *Maximize Common Value* is its dual. At this writing, there are about 60 Advisors, with more under development. The next section sketches a variety of ways that FORR supports ACE's autonomous enhancement of its decision process.

Reformulating the Decision Process

FORR computes a variety of *meta-heuristics* (knowledge about its heuristics) on its Advisors to support the reformulation of the decision process (Epstein, 2004). The most basic reformulation is the weight learning in tier 3, which uses meta-heuristics for accuracy and risk to reward and punish the Advisors. A FORR-based program learns on a set of examples from its problem class, and then is tested on a different set of examples from the same class. 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. As a result, only that useful knowledge referenced by the remaining Advisors need be computed during testing.

A FORR-based program can also learn new tier-3 Advisors. Learning new Advisors requires a programmer-specified language in which to express them. (Details in (Epstein, Gelfand & Lock, 1998).) FORR monitors these expressions; good ones eventually participate in the decision process as contributors to a single Advisor that represents the language. The very best become individual Advisors. For example, ACE has a language that considers products and quotients of four metrics during three stages of problem solving. On <30, 8, .1,. 5>, a class of relatively small problems, ACE consistently learned an individual heuristic this way, a novel one to the CSP community. ACE's learned heuristic was incorporated into three different, traditional CSP solvers, outside of ACE. The originals and their enhanced versions were tested on reasonably difficult, much larger problems from <150, 5, .05, .24>. The enhanced versions searched 25% – 96% fewer nodes than the original ones (Epstein, et al., 2002).

Another reformulation is the identification of multiple stages during a problem's solution, stages during which tier-3 Advisors have different weights. ACE learns, as useful knowledge, the location of a break point for a final stage. After that break, and only when selecting a variable during testing, ACE supersedes tier 3, applying the highest-weighted variable-selection Advisor alone, before the others. The break effectively creates a second stage where most weights are zero, and usually speeds decision making without introducing additional error. For relatively easy problems, ACE learns that, in the final stage, it is actually more efficient not to reason at all, and selects variables there arbitrarily.

Two structural reformulations are also available in FORR: promotion and prioritization. *Promotion* moves into tier 1 any tier-3 Advisor whose weight is so high that the Advisor appears correct. Promotion speeds computation because the decision is either made in tier 1 or fewer alternatives survive for consideration in tier 3. This speedup, however, is rarely worth the extended search that results from any promoted Advisor's occasional egregious errors. *Prioritization* partitions those tier-3 Advisors retained for testing into a hierarchy of subsets based on their learned weights. Under prioritization, the top-ranked subset

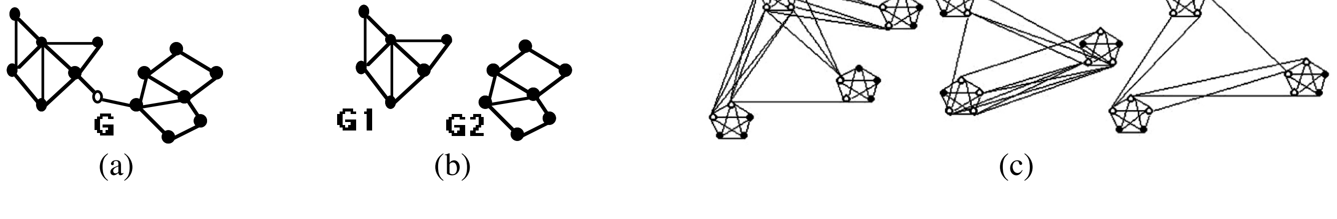

Figure 4: DSO for ACE is inspired by spatial cognition. A constraint graph (a) before and (b) after the variable at the white vertex is assigned a value. Assignment removes the vertex and its edges, producing two connected components, each of which could become a subgoal. (c) Some constraint graphs for logic puzzles, which benefit from such an approach.

votes first; if it determines a best action, that becomes the decision. If a subset prefers more than one action (a tie), 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. Prioritization speeds computation because a decision can often be made without devoting cycles to the full complement of tier-3 Advisors. Under prioritization into some (problem-class-dependent) number of subsets, ACE typically reduces its computation time by 50% or more, without sacrificing accuracy.

There is a distinction in FORR between the programmer, who writes domain-specific code, and the user, who merely executes experiments using that code. The user, however, has considerable latitude in the design of an experiment as a sequence of problem solution attempts called *phases*. Beyond the ordinary "learn and then test," an experiment may include a *preliminary phase* to test and then suppress poor Advisors before learning to solve problems. It may have multiple learning phases to support learning transfer, where a program learns first on an easier class and then continues learning on a harder class. It may include an analysis phase to reformulate the decision structure before testing (or before additional learning). It is also possible to ablate Advisors, tiers, and even types of phases entirely. There are also general options (e.g., fast and frugal reasoning methods). One option is the ability to halt when learning no longer has an impact. For example, as learning progresses, corrections under one of the weight-learning algorithms become smaller, relative to the overall weight. In this case, corrections have a diminishing impact, so a program can monitor the weight fluctuations of its tier-3 Advisors, and determine on its own when to stop learning.

Related and Current Work

Because FORR-based programs do unsupervised learning through trial and error with delayed rewards, they are reinforcement learners (Sutton & Barto, 1998). Ordinarily, reinforcement learning learns a *policy*, a mapping from estimated values of repeatedly experienced states to actions. In learning to solve a broad class of challenging problems, however, one is unlikely ever to revisit a state, given the size of both an individual problem's search space and the size of a problem class. Instead, FORR learns a policy that tells the program how to act in any state, one that *combines* action preferences as expressed by its Advisors' comments.

Two AI artifacts are reminiscent of FORR: STAGGER and SAGE.2. STAGGER could learn new heuristics, but its learning 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 (Schlimmer & Fisher, 1986). SAGE.2 also learned search control from unsupervised experience, reinforced decisions on a successful path, gradually introduced new factors, specified a threshold, and could transfer its ability to harder problems (Langley, 1985). SAGE.2, however, learned repeatedly on the same task, reinforcing repeating symbolic rules, while ACE learns on different problems in a specified class, reinforcing the sources of correct comments. When SAGE.2 failed, it revised its rules uniformly; when ACE errs, it reduces its weights in proportion to the size of the error. SAGE.2 learned during search, and compared states, but it lacked random benchmarks and subgoals. ACE, in contrast, learns only after search and does not compare states, although it has both random benchmarks and subgoals.

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). 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, which the user had to represent 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.

Current work on FORR includes a learned structure for tier 2, the application of additional meta-heuristics to identify prioritization classes, and new weight-learning algorithms. Finally, if ACE engineers a problem-solving paradigm, then it is both frugal and clever to specialize ACE itself for particular kinds of CSPs. For example, *Later* is a tier-1 Advisor that delays the selection of any variable whose dynamic domain is larger than its forward degree. Later is incorrect for general CSPs, but correct for graph coloring. Using Later, ACE rediscovered the well-known Brélaz heuristic (Epstein & Freuder, 2001). There are Advisors and useful knowledge specific to other kinds of CSPs as well. Adding them to ACE would both benefit from the CSP paradigm and enhance it.

FORR-based programs are intended as colleagues in research. ACE, for example, can support constraint programmers in their quest for method combination appropriate to a particular class of problems specified by the user. It can also support a novice constraint programmer in the selection of heuristics. ACE can learn new, efficient heuristics, ones that were previously unidentified by experts and can be readily used by them in other programming environments. It can learn heuristics for problem classes that do not succumb to ordinary, off-the-shelf CSP approaches. Thus we do not pit a FORR-based program against others, but use it to provide insight, and to export problem-solving improvements to other solvers, both human and machine.

Acknowledgements

This work was supported in part by NSF IIS-0328743 and by Science Foundation Ireland under Grant 00/PI.1/C075. Thanks to CUNY's FORR study group and the Cork Constraint Computation Centre, led by Gene Freuder, for their

continued support of this work.

References

Agre, P. E. & D. Chapman (1990). What are Plans for? *Robotics and Autonomous Systems* 6: 17-34.

Beck, J. C., P. Prosser & E. Selensky (2003). Vehicle routing and job shop scheduling: What's the difference? *Proceedings of the 13th Int'l Conference on Automated Planning and Scheduling -ICAPS03* pp. 267-276.

Biswas, G., S. Goldman, D. Fisher, B. Bhuva & G. Glewwe (1995). Assessing Design Activity in Complex CMOS Circuit Design in *Cognitively Diagnostic Assessment*, ed. P. Nichols, S. Chipman and R. Brennan. Hillsdale, NJ, Lawrence Erlbaum pp. 167-188.

Cheeseman, P., B. Kanefsky & W. M. Taylor (1991). Where the REALLY Hard Problems Are. *IJCAI-91* pp. 331-337.

Crowley, K. & R. S. Siegler (1993). Flexible Strategy Use in Young Children's Tic-Tac-Toe. *Cognitive Science* 17: 531-561.

D'Andrade, R. G. (1990). Some Propositions about the Relations between Culture and Human Cognition in *Cultural Psychology: Essays on Comparative Human Development*, ed. J. W. Stigler, R. A. Shweder and G. Herdt. Cambridge, Cambridge University Press pp. 65-129.

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

Epstein, S. L. (1998). Pragmatic Navigation: Reactivity, Heuristics, and Search. *Artificial Intelligence* 100: 275-322.

Epstein, S. L. (2001). Learning to Play Expertly: A Tutorial on Hoyle in *Machines That Learn to Play Games*, ed. J. Fürnkranz and M. Kubat. Huntington, NY, Nova Science pp. 153-178.

Epstein, S. L. (2004). Metaknowledge for Autonomous Systems. *Proc. AAAI Spring Symposium on Knowledge Representation and Ontology for Autonomous Systems*, pp. 61-68.

Epstein, S. L., E. C. Freuder, R. Wallace, A. Morozov & B. Samuels (2002). The Adaptive Constraint Engine in *Principles and Practice of Constraint Programming -- CP2002*, LNCS 2470, ed. P. Van Hentenryck. Berlin, Springer Verlag pp. 525-540.

Epstein, S. L. & G. Freuder (2001). Collaborative Learning for Constraint Solving in *CP2001*, 2239, ed. T. Walsh. Berlin, Springer Verlag pp. 46-60.

Epstein, S. L., J. Gelfand & E. T. Lock (1998). Learning Game-Specific Spatially-Oriented Heuristics. *Constraints* 3: 239-253.

Epstein, S. L. & T. Ligorio (2004). Fast and Frugal Reasoning Enhances a Solver for Really Hard Problems. *Proceedings of Cognitive Science 2004.* In press.

Ericsson, K. A. & J. Staszewski (1989). Skilled Memory and Expertise: Mechanisms of Exceptional Performance in *Complex Information Processing: The Impact of Herbert A. Simon*, ed. D. Klahr and Kotovsky. Hillsdale, NJ, Erlbaum pp. 235-267.

Freuder, E. & A. Mackworth (1992). *Constraint-Based Reasoning.* Cambridge, MA, MIT Press.

Gent, I. E., E. MacIntyre, P. Prosser & T. Walsh (1996). The Constrainedness of Search. *AAAI-96,* pp. 246-252.

Gigerenzer, G. & P. M. Todd (1999). *Simple Heuristics that Make Us Smart*. New York, Oxford University Press.

Keim, G. A., N. M. Shazeer, M. L. Littman, S. Agarwal, C. M. Cheves, J. Fitzgerald, J. Grosland, F. Jiang, S. Pollard & K. Weinmeister (1999). PROVERB: The Probabilistic Cruciverbalist. *AAAI-99,* pp. 710-717.

Klein, G. S. & R. Calderwood (1991). Decision Models: Some Lessons from the Field. *IEEE Transactions on Systems, Man, and Cybernetics* 21: 1018-1026.

Langley, P. (1985). Learning to Search: From Weak Methods to Domain-Specific Heuristics. *Cognitive Science* 9: 217-260.

Lock, E. & S. L. Epstein (2004). Learning and Applying Competitive Strategies. *AAAI-04* . In press.

Minton, S. (1996). Automatically Configuring Constraint Satisfaction Programs: A Case Study. *Constraints* 1: 7-43.

Nichelli, P., J. Grafman, P. Pietrini, D. Alway, J. Carton & R. Miletich (1994). Brain Activity in Chess Playing. *Nature* 369: 191.

Ratterman, M. J. & S. L. Epstein (1995). Skilled like a Person: A Comparison of Human and Computer Game Playing. *Proc. Cognitive Science 1995*, pp. 709-714.

Sacks, O. (1985). *The Man Who Mistook his Wife for a Hat.* Summit (New York), Simon & Schuster.

Schlimmer, J. G. & D. Fisher (1986). A Case Study of Incremental Concept Induction. *AAAI-86,* pp. 496-501.

Schraagen, J. M. (1993). How Experts Solve a Novel Problem in Experimental Design. *Cognitive Science* 17: 285-309.

Siegler, R. S. & K. Crowley (1994). Constraints on Learning in Nonprivileged Domains. *Cognitive Psychology* 27: 194-226.

Simon, H. A. (1981). *The Sciences of the Artificial*. Cambridge, MA, MIT Press.

Sutton, R. S. & A. G. Barto (1998). *Reinforcement Learning: An Introduction*. Cambridge, MA, MIT Press.

Tsang, E. P. K. (1993). *Foundations of Constraint Satisfaction*. London, Academic Press.

Wallace, M. (1996). Practical Applications of Constraint Programming. *Constraints*, *1*, 139-168.

Yoshikawa, A., T. Kojima & Y. Saito (1998). Relation between the Way of Using Terms and the Skill - from Analysis of Protocols of the Game of Go. *Proc.1st Int'l Conference on Computers and Games* pp. 211-227.