Metaknowledge for Autonomous Systems

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

Abstract

An autonomous system is postulated here as a collection of cooperating heuristics. The goal of such a system is to become expert in a particular domain by solving problems there. The system develops by analyzing the performance of its heuristics, and changing its decision process to reflect its knowledge about them. Metaknowledge metrics are postulated both to evaluate the system's developing expertise and to evaluate the heuristics on which it is based. The implementation of these metrics within a problem-solving architecture is discussed, and their impact on an application that learns to solve challenging, large-scale problems is detailed.

Introduction

A *discovery system* is expected to learn autonomously from its problem-solving experience. Such a system probes its environment for data from which to learn. Without human intervention, the system acquires knowledge and, ideally, becomes an expert in its domain. The thesis of this work is that such a system can develop substantial expertise as it solves problems and analyzes its own components and performance. The principal contribution of this paper is a metaknowledge ontology for systems that use multiple representations, multiple learning methods, and multiple heuristics. It begins by outlining an approach which decouples representation and learning from reasoning. The paper then discusses metaknowledge for heuristics and for self-evaluation. The final section describes a learning and problem solving architecture that derives its power from many of these ideas and expects to benefit from the others shortly.

Fundamental Assumptions

The autonomous systems under consideration here have as their task the achievement of expert performance in a particular domain, where performance is the ability to solve problems. In this paper, a *problem* is an initial state of the world, a set of actions that can be applied to transform one world state into another, and a goal test. The goal test accepts a world state and returns a numerical value when a problem is solved. The returned value measures how skillful the solution process was; it may therefore incorporate such factors as execution time and number of retracted erroneous decisions. *Problem solving* is thus finding a sequence of actions that transforms the initial state into a positive response from the goal test. *Expert problem solving* finds a sequence of actions that receives a high-valued response from the goal test.

Because an autonomous system cannot rely upon human intervention, it must be able to learn. In an unknown or dynamic environment, all world states are not predictable, and any pre-organized set of production rules is likely to be brittle. A prudent designer therefore provides a variety of tools with which to consider alternative actions. These include representations, learning methods, and general heuristics that harness multiple perspectives on decision making to solve problems. This approach has succeeded in several domains, including game playing (Epstein 2001), path finding (Epstein 1998), challenging crossword puzzles (Keim et al. 1999), and constraint satisfaction (Epstein et al. 2002). Some of these systems can also learn new heuristics that are subsequently incorporated in the decision process (Epstein et al. 1998). Whether its heuristics are pre-specified or learned, the system should trust them only as they prove helpful in problem solving.

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 game board might be represented as a list, as a two-dimensional layout, and also as lines of attack.) When more than one heuristic employs the same representation, it is possible to compute the representation only once and store it, so that it is readily available to any other heuristics that may use it. In this way, an autonomous system need not credit or blame a representation, only the heuristics that reference it. Furthermore, if no successful heuristic ever employs a particular representation, the system can eventually note that, drop the representation, and redirect the computational resources once devoted to it.

Similarly, it is possible to decouple learning methods from the reasoning process. An autonomous system should acquire only knowledge that it can subsequently apply. The system's skill lies not in the agglomeration of information but in the intelligent application of that knowledge once it is acquired. Knowledge is applied in decision making by a process (here, a heuristic) that references it. Thus an autonomous system need not credit or blame a particular learning method, only the heuristics that reference the knowledge it acquires. Indeed, if no

successful heuristic ever employs knowledge derived from a particular method, then the system can eventually note that and drop both the method and the knowledge it produces, again conserving computational resources.

Knowledge about Performance

A human expert is one who performs a task faster and better than the rest of us (D'Andrade 1990). Because the autonomous learner considered here is intended to develop into an expert, it should monitor both its speed and its performance. Speed is captured by CPU seconds to solution, but performance requires a variety of measures. Possibilities include number of problems solved (if the system is not complete), number of retracted decisions (*errors*) during solution, and number of errors relative to the size of the problem.

As an autonomous system *ages* (operates alone across time), the quality of its performance is likely to vary. The system may make decisions differently, or a dynamic environment may present different kinds of problems. In response, an autonomous system can seek to improve the speed with which it solves problems, or seek out more difficult ones.

The *consistency* of a system's expertise reflects its ability to solve problems. It can be measured by the number of consecutive problems solved or by the percentage of problems solved within some recent window. As a system ages, it should not only become more consistent and faster, but should also improve its *power*, its ability to solve problems without any errors at all.

An autonomous system should be able to restart as necessary. If search for a solution to a particular problem is not proceeding well, the system should be able to choose to *restart* the problem, that is, to begin to solve it once again, but with a different initial decision. Repeated restarts of the same problem with an increasing resource bound help the system determine the degree of difficulty of a problem. Once a problem is solved, repeated restarts with a decreasing resource bound enable the system challenge itself to perform better, based on what it has learned. In addition, if the system judges its performance on a particular set of problems to be poor, the system should be able to choose to restart the entire process, that is, to start afresh on the current problem set.

Heuristics and Planners

For our purposes, a *heuristic* is a decision process that, when *consulted* (given the current state of the problem-solving world), may *comment* (express an opinion about which action to take). A comment may be expressed either as a suggestion (e.g., "do *x*") or as a preference (e.g., "*x* is better than *y*"). More generally, comments may include alternative actions (e.g., "do *x* or *y*"), oppose an action (e.g., "don't do *z*"), or value preferences and aversions with a numerical *strength* (e.g., "*x* is a 10, *y* is an 8, but *z* is a –3").

A heuristic whose comment references more than one action in combination (e.g., "do *x* and *y*") is here termed a *planner*. Its comment (*plan*) may be ordered, partially-ordered, or merely a set of actions which it believes should take priority over all other alternatives. With this characterization, planners become heuristics whose comments may need other heuristics to assist in their execution. For example, if a planner specifies an unordered set of actions, the sequence in which they are executed falls to the single-action heuristics, which then comment upon them as if they were a restricted set of possible actions.

Heuristics, either pre-specified or learned, are presumably present in an autonomous system because they are expected to be relevant to the domain of interest. (For example, avoiding dead-ends is relevant to path finding, while emphasizing good openings is relevant to game playing.) Heuristics, however, come with no guarantee. An autonomous learner should therefore winnow through its heuristics, discarding some and emphasizing others. There is, in addition, no reason to assume that heuristics are independent of each other. To manage a collection of questionable, potentially-dependent heuristics successfully requires a host of evaluation metrics, discussed individually below.

Knowledge about Heuristics

The computation of any metric on heuristics should consider *timing*, the position an action occupies in a solution sequence. There is no reason to suppose that a heuristic behaves uniformly across a solution sequence. (For example, a heuristic that provides good openings for game playing will not be equally prized in the final moves of a contest.) A simple, workable approach is to partition each solution sequence into *stages* (contiguous problem-solving intervals, e.g., "early," "middle," and "late") and then evaluate a metric separately within each stage.

An autonomous system should evaluate its heuristics to improve its expertise. Low-performing heuristics can be discarded, and high-performing ones can in some manner be emphasized, so that the system makes better decisions and makes them more quickly. These decisions should be made within the context of timing, that is, each metric should be computed for each heuristic by stage. The following metrics on heuristics are proposed.

Accuracy. The *accuracy* of a heuristic is the frequency with which it makes correct decisions. If an autonomous system can judge whether or not, in retrospect, a decision was correct, it can tabulate the accuracy of each of its heuristics as it solves problems. There are a variety of

ways to tabulate accuracy, including number of correct comments and percentage of correct comments.

It may not be possible to specify an absolute standard of heuristic accuracy before problem solving begins. This is because an autonomous decision maker is repeatedly faced with sets of choices, whose size and nature will vary. In this case, a useful device with which to gauge accuracy is a *benchmark*, a baseline procedure for a heuristic that faces the same decisions as its heuristic, but constructs random comments on randomly many actions, with random strengths. A heuristic whose accuracy is lower than its benchmark during a particular stage can justifiably be ignored, since that heuristic makes no better contribution to expertise than a random process would.

Stability. The *stability* of a heuristic is the consistency of the heuristic's accuracy across the system's lifetime. A variety of statistical methods gauge stability, including the change in standard deviation of accuracy across time, or moving averages on accuracy. Whatever method is used to compute stability, it is important not to change the system's treatment of a heuristic until both the heuristic and its benchmark are deemed stable.

Utility. Two heuristics may be equally accurate in their decisions, but differ dramatically in their resource consumption. The *utility* of a heuristic is the ratio of its accuracy to its cost. (Cost is measured in time here, plus space if a system is memory-constrained.) The cost of computing a heuristic could conceivably outweigh the benefit the system derives from its comments, particularly if the heuristic scores low on several other metrics. At the very least, a low-utility heuristic in a stage should probably be reserved for decision situations where other, equally accurate heuristics have been unable to finalize a decision.

Influence. A heuristic may be unnecessary, even though it is accurate and has high utility. A heuristic *drove* a decision if and only if a different action would have been chosen without that heuristic's comments. The *influence* of a heuristic is the frequency with which it drives decisions when it comments. Regardless of its utility, any heuristic with low influence in a stage makes little contribution to the decision process, and is a candidate for elimination. Some caution must be exercised here, however. For example, if two ostensibly different, highly-accurate heuristics are highly correlated, neither might have high influence, but the removal of them both could reduce accuracy.

Novelty. The *novelty* of a heuristic compares its applicability to that of other heuristics. A heuristic that often comments on some action on which others have no opinion is *novel*; one that comments only in concert with many others is not. A novel, accurate heuristic within a stage is worth retaining.

Acuity. The *acuity* of a heuristic is the degree to which it is capable of discriminating among alternatives. A heuristic that singles out relatively few actions as worthy is more valuable than one that finds almost all actions equally acceptable. A heuristic that assigns different strengths to a set of actions is more valuable than one that supports or opposes them all equally. An acute, accurate heuristic within a stage is worth retaining if no heuristic surpasses it on other metrics.

Involvement. The frequency with which a heuristic comments is its *involvement*. To comment, a heuristic must be able to distinguish among alternative actions, that is, it must not award every action the same strength. A heuristic with low involvement may still be valuable in a stage, particularly if its acuity or novelty is high.

Risk. The *risk* associated with a heuristic is the expected computational cost incurred if it were the sole decision maker. Risk should be computed from the likelihood of error and the additional work that error would incur. This metric becomes important when the system attempts to accelerate decision making without sacrificing performance quality. A low-risk, highly-accurate heuristic may warrant special treatment during decision making. (See the discussion of promotion and prioritization below.)

Before any heuristic participates fully in decision making, it must comment often enough for the metrics to evaluate it properly. Until then, a heuristic should have probationary status, during which its comments are *discounted* (considered less important) during decision making. Discounting preserves any consistency the system has already developed, particularly if the system learns new heuristics. After the discount period, the decision computation should take the metrics into careful consideration, allowing valuable heuristics their full impact, and preventing poor heuristics from doing harm.

A Demonstration: FORR and ACE

Many large-scale, real-world problems are readily represented, solved, and understood as *constraint satisfaction problems* (CSPs). A *CSP* is 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 here to binary CSPs, whose constraints involve at most two variables.) A value assignment for all the variables is a *solution* for a CSP if and only if it satisfies all the constraints. Every CSP has an underlying *constraint graph*, that represents each variable by a vertex whose possible labels are its domain values. An edge in a 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

```
Until the problem is solved
  Select an unvalued variable v
  Assign v the value a
  If v = a is consistent with all currently
     assigned values
  then continue
  else backtrack to the last variable for
    which there was an alternative value
```

Figure 1. An algorithm to solve a CSP.

simple CSP solution algorithm appears in Figure 1.

FORR (For the Right Reasons) is an architecture for learning and problem solving that supports the development of expertise (Epstein 1994). FORR relies on a collection of domain-specific heuristics to solve problems, and supports learning new heuristics. FORR-based programs have developed considerable expertise in game playing, path finding, and constraint satisfaction (Epstein 1998; Epstein 2001; Epstein et al. 2002). Throughout this section, examples will be given from *ACE* (the Adaptive Constraint Engine), a FORR-based system for constraint solving (Epstein et al. 2002). ACE is an autonomous system that *learns* to solve CSPs. It is an ambitious program — armed with dozens or even hundreds of heuristics, it can solve difficult problems. As such, it is an excellent domain within which to demonstrate FORR, and to explore the behavior of collections of heuristics.

CSP solution is NP-complete. In response, CSP researchers have produced a wealth of good, general-purpose heuristics to solve a broad range of real-world problems (Nudel 1983; Freuder et al. 1994). The solution of a CSP remains more art form than automated process, however, in part because the interactions among existing heuristics are not well understood. Thus each new, large-scale CSP faces the same bottleneck: difficult constraint programming problems need people to "tune" a solver efficiently. ACE addresses that bottleneck by learning to be an expert on a set of problems, autonomously.

A Hierarchy for Heuristics

To use FORR, one defines a domain and a set of relevant heuristics. The heuristics are initially classified into a hierarchy of three *tiers* by the user; FORR moves through them to make a decision. As Figure 2 indicates, the heuristics in tier 1 are consulted sequentially; the heuristics in tiers 2 and in tier 3 are consulted (effectively) in parallel.

Tier 1 consists of *perfect* (i.e., error-free) heuristics consulted sequentially. If any heuristic comments in favor of an action, that action is executed, without reference to any subsequent heuristics. Consulting perfect heuristics first ensures that obvious correct decisions (e.g., capturing a king in chess) are reached without devoting resources to other, less reliable heuristics. A perfect heuristic that comments against an action (e.g., "don't move into checkmate") removes that action from consideration by all subsequent heuristics, thereby preventing obvious errors. Placing perfect heuristics in tier 1 permits easy problems to be solved easily, a feature all too rare in complex systems.

Tier 1 heuristics are generally common knowledge to domain experts, and are quick to compute. A simple example of a tier-1 heuristic from ACE is Victory. If only a single variable remains without an assigned value, and one or more consistent values are available, Victory

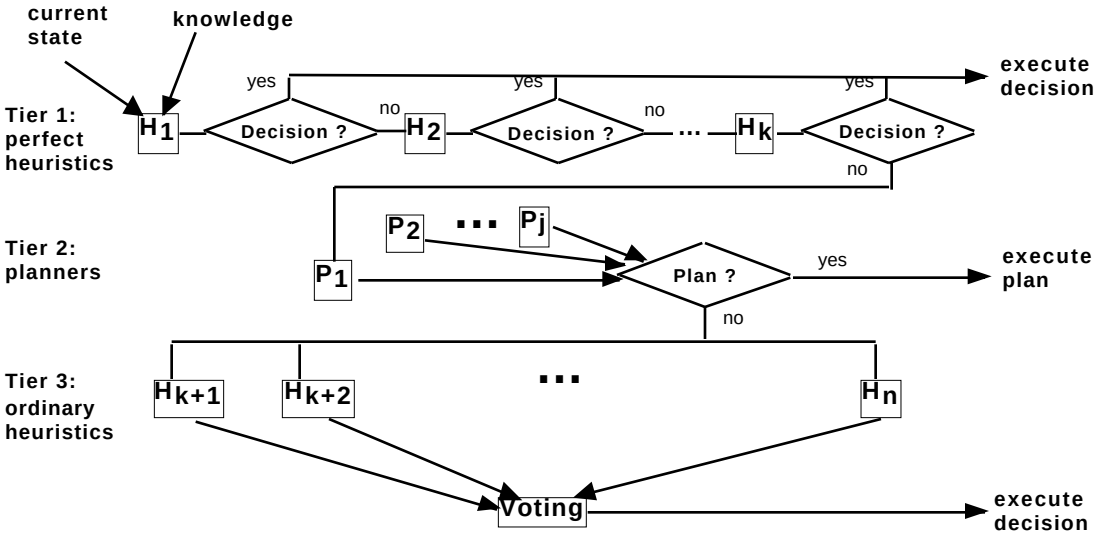

Figure 2: The FORR decision hierarchy. Heuristics (in boxes) in tier 1 either determine an action or forward a (possibly reduced) set of choices. Heuristics in tier 2 identify a subset of choices that take priority in subsequent decisions. Heuristics ir tier 3 determine an action collectively.

comments to make an arbitrary assignment.

FORR's *tier 2* organizes the planners for a domain. A plan, recall, is a set of actions. Planners are intended to be quick and heuristic. Plans in FORR are abandoned (rather than repaired) when their actions are no longer executable during solution, so limited resources are devoted to plan construction. Locating planners in tier 2 both retains the correctness of the tier-1 heuristics and values longer-range perspectives over the single-action heuristics of tier 3. A heuristic named Enforcer, early in tier 1, guarantees that the actions in a plan produced by tier 2 take priority over other actions, until the plan is either completed or becomes inapplicable and is therefore abandoned.

ACE's current planners capitalize on the CSP graph. Subgraphs are identified that should be solved as a whole, before other segments of the problem are addressed, or should be postponed until the remainder of the problem is completed. Some planners may be perfect, such as ACE's No Tree, which postpones consideration of entire acyclic components. Others are heuristic, such as ACE's Maximally Restricted, which addresses the most constrained connected component first.

Tier-3 heuristics in FORR produce single-action comments that are not guaranteed to be correct. These generally form the bulk of a FORR-based system. Because they are fallible, their comments are combined to select the next action in a process called *voting*.

ACE's tier-3 heuristics were, for the most part, drawn from the CSP literature. In every case, however, the dual of the heuristic's underlying metric was also implemented. For example, the CSP literature suggests that the next variable to have a value assigned to it should have a minimum *dynamic domain size* (set of values that would still be consistent with existing variable assignments). ACE therefore has a heuristic that comments in favor of such variables, but it also has its dual, a heuristic to maximize the dynamic domain size of a variable. ACE has two benchmarks for accuracy: one for heuristics that select a variable, and one for heuristics that select a value.

Implementing Metaknowledge

FORR now implements most of the ideas presented earlier in this paper. Both representation and learning methods are decoupled from heuristics. The current world state offers multiple representations, and every heuristic has equal access to them. Heuristics are permitted to make multiple comments that support or oppose actions, and assign strengths to those opinions.

A FORR-based system (e.g., ACE) runs in phases, with learning followed by testing. During a *learning phase*, metaknowledge is acquired. Between learning and testing, metaknowledge is applied to evaluate and possibly revise the components of Figure 2. Then, during each *testing phase*, learning is turned off and performance is measured. The solution of an individual problem is partitioned into stages pre-specified by the user. For example, in ACE one may specify the early stage of problem solving as the first 20% of the variable assignments, the late stage as the last 30%, and relegate the remainder of the decisions to a middle stage. During learning, FORR gathers metaknowledge about each tier-3 heuristic for each stage.

FORR monitors the number of steps required to solve a problem, and (in domains where it is appropriate) the number of retracted decisions made during solution. Problem solving may be limited by the user to some number of steps or seconds of CPU time. FORR supports both individual problem restart and restart on an entire learning phase. The ability to restart a problem, the number of restarts permitted, and increasing resource bounds are specified by the user.

FORR can terminate learning based on a variety of criteria. The user can specify that learning should stop after some number of problems, enough to ensure (based on previous observation of the system) that expertise has leveled off. It is also possible, however, to have the system decide that it has learned enough. It can terminate learning when some consecutive number of expert solutions (those within some number of decisions) have been reached, or when all heuristics and their respective benchmarks are stable. Still another option permits the entire process to restart, based upon some number of failures to solve problems during learning.

FORR measures the accuracy of a heuristic by its *weight*. There are a variety of weight-learning algorithms, all of which include discounting. One algorithm, for example, tabulates the ratio of the number of expert-like comments to the number of comments (Epstein 2001). For domains such as CSP, where an error and its severity can be readily identified by the system, more sophisticated weight-learning algorithms blame the heuristics responsible for each error (Epstein et al. 2001). Poor heuristics, those whose accuracy is lower than that of their respective benchmarks, are eliminated from decision making on a stage by stage basis, when learning is over, but before testing begins.

Experimental Results

A CSP can be characterized by four parameters: its number of variables *n*, its maximum domain size *k*, its *density d* (the fraction of possible edges it includes), and its *tightness t* (the percentage of possible value pairs it excludes). A *problem class* is a set of CSPs with the same parameters. In the descriptions that follow, a class is described as *n-k-d-t*, for example, 30-8-.1-.5 is the class of problems on 30 variables with maximum domain size 8, density .1, and tightness .4.

A problem in *n-k-d-t* presents a substantial search tree, potentially of size $O(k^n)$. Although CSP solution is NP-hard, some problem classes surrender readily to heuristics. A problem with low density and tightness will be relatively easy to solve and usually admit multiple solutions, while one with high density and tightness is likely to have no solutions at all. The most difficult problem classes, for a fixed number of variables and fixed domain size, generally lie within a relatively narrow range of pairs of values of density and tightness (Cheeseman et al. 1991).

ACE solves a CSP by repeatedly selecting a variable and then selecting a value for it, as in the algorithm of Figure 1. In the experiments reported here, ACE used MAC3 (maintained arc consistency) to constrain the domains of the neighbors (in the constraint graph) of each newly-valued variable, extending those restrictions repeatedly to every unassigned variable until no variable's domain changed. Retractions were done with standard backtracking.

In the empirical results that follow, each experiment was performed on a set of randomly-generated problems for a single class. Learning on the first 600 problems was followed by testing on 200 new problems. (Typically such a problem set is produced by a random generator. Given the enormity of the problem space, the likelihood of generating the same problem more than once in a set of 800 problems is negligible.) No planners were employed, and the system was not permitted to learn new heuristics.

A variety of metrics may be applied to gauge the performance of a CSP solver: computation time, *constraint checks* (times that pairs of values are confirmed acceptable to a constraint), *retractions* (times that a value selection or a variable selection is withdrawn during backtracking), and *power* (percentage of retraction-free solutions). All cited differences are statistically significant at the 95% confidence level within the cited problem class, but may not pertain to all CSPs.

Results with Metaknowledge

The following examples of the power of metaknowledge are all drawn from ACE:

• Eliminating inaccurate heuristics provides substantial speedup during testing, typically by a factor of 2. In 30-8-.12-.5 problems, for example, the mean execution time is reduced by 55% when only heuristics with weights higher than their respective benchmarks' are consulted.

• Measuring accuracy in graph coloring problems (a subset of CSP), ACE quickly discarded many of its input heuristics, some of which had been deliberately included to mislead the program. In the process, ACE independently rediscovered the Brélaz heuristic, a well-known theorem in graph coloring (Epstein et al. 2001).

• While learning new heuristics for CSP solution, ACE identified an outstanding, hitherto unknown heuristic for early decisions only. When this heuristic was exported to a conventional CSP solver, it improved performance on far more difficult problems by as much as 96% (Epstein et al. 2002).

The aforementioned duals of popular heuristics have, in certain stages and problem classes, proved themselves valuable. For example, a popular CSP heuristic for selecting the next variable to bind is "minimize the ratio of the dynamic domain size of a variable to its original degree in the underlying constraint graph." In most problem classes, ACE judges this Min-Domain/Degree heuristic to be highly accurate. In 30-8-.05-.5 problems (a relatively easy class), however, the Max-Domain/Degree heuristic achieves a far higher weight during the middle of problem solving (variables 7 through 24) than the traditional Min-Domain/Degree.

Indeed, distinct problem classes appear to have heuristic *signatures*, that is, respond best to different combinations of heuristics. For example, on 30-8-.08-.5 problems and 30-8-.1-.5 problems, ACE identifies the same set of variable-selection heuristics as accurate, but the value selection heuristics are duals of each other, that is, when a maximizing heuristic is accurate in one class, the minimizing version is accurate in the other.

Metaknowledge under Development

Work is in progress on FORR to calculate and apply a variety of metaknowledge:

• FORR tracks which heuristics access which representations. At the moment, all representations are retained, but as FORR winnows out heuristics, unreferenced representations will no longer be computed.

• The delineation of stages will ultimately be learned. Recall that, currently, stage boundaries are specified by the user. Initial testing has indicated that multiple stages can provide valuable guidance, but that weight learning on too many stages may overfit the available data. For example, with a single stage ACE learns to solve 30-8-.1-.5 problems in less than half the time that is required when it uses 30 stages. Current work includes having FORR learn stage boundaries for the accuracy of each heuristic individually. Those boundaries will in turn be used for the computation of other metaknowledge. One way FORR could generate stage boundaries would be to subdivide the solution sequence when the quality of a heuristic varies broadly within it. Another approach would be to begin with many short stages and have the system coalesce adjacent ones where a metric has similar reported values.

• For each tier-3 heuristic and each stage, FORR gauges stability as the standard deviation of the heuristic's weight in that stage over the most recent tasks. (Current settings are a standard deviation of less than 0.1 over the last 40 tasks.) Preliminary results indicate that terminating learning once all heuristics reach stability shortens learning time and does not impact performance.

• FORR now collects, for each tier-3 heuristic by stage, the full panoply of metaknowledge: accuracy, stability, utility, influence, novelty, acuity, involvement, and risk. Risk is currently being tested as a factor in the penalty assignment during weight learning.

Metaknowledge for Autonomous Restructuring

Self-awareness with respect to heuristics can support autonomous reformulation of the program's decision-making structure. We offer two examples from FORR here: promotion and prioritization. Recall that the system designer currently assigns a heuristic to a tier. A heuristic that is known by people to be correct is placed in tier 1, a planner is placed in tier 2, and all other heuristics are relegated to tier 3.

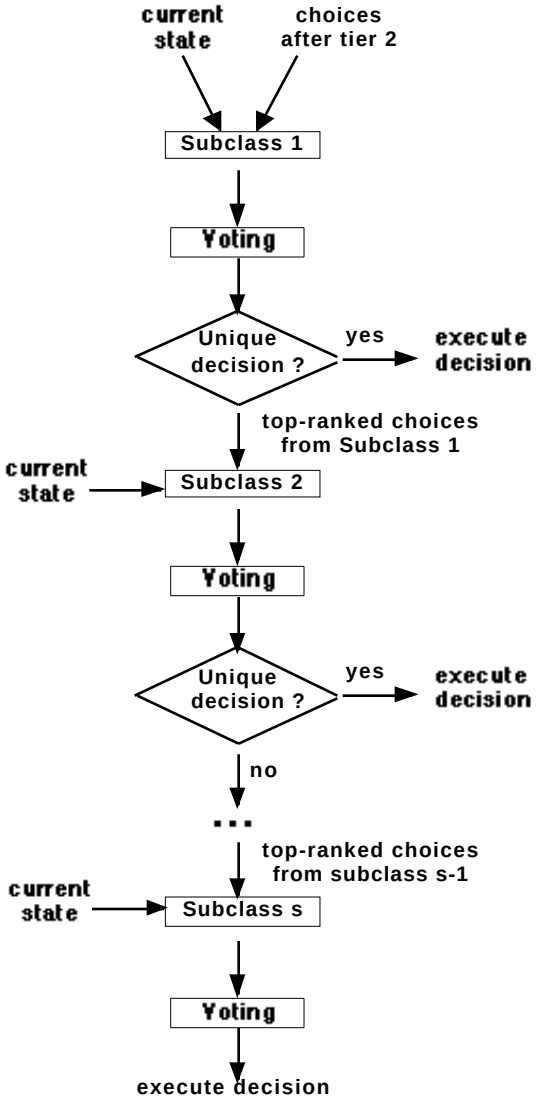

Figure 3. Restructuring tier 3 into *s* subclasses under prioritization.

Promotion is the advancement to tier 1 of a heuristic that is *nearly perfect* (almost always correct) for a given stage. (In some domains, such as game playing, promotion is unacceptable, because even the smallest likelihood of error is intolerable. Happily, this is not the case with CSPs, where an incorrect choice can always be retracted.) An example of a potentially promotion-worthy CSP heuristic is Later, which opposes the selection of any variable whose dynamic domain is larger than its degree. For graph coloring, Later is provably correct and belongs in tier 1. For general CSPs, however, Later is occasionally wrong, and can impact performance dramatically. For example, adding Later to tier 1 reduces ACE's execution time on 20-8-.14-.5 problems and increases its power. On 30-8-.12-.5 problems, however, Later has quite a different effect: its presence even in tier 3 forces ACE to do more constraint checks and to lose nearly half its power. We intend to have ACE learn to promote heuristics in accordance with metaknowledge, on a class by class basis.

Prioritization is the stratification of tier 3 into *s* subclasses for a given stage, so that the best of the ordinary heuristics can take priority, as a group, over the others. A diagram of tier-3 decision making under prioritization appears in Figure 3. The number of subclasses *s* should be determined by the system, as well as the grounds on which to partition them. Currently, with prioritization, FORR uses accuracy as the only discriminator, and retests decision making 8 times on the same problems, with *c* subclasses, where $c = 1, 2, \ldots, 8$. Prioritization into 3 subclasses enabled ACE to solve 10-8-.1-.5 problems in less than half the time, but more subclasses did not improve performance. On 30-8-.1-.5 problems, however, 5 subclasses produced the best performance, while more than 5 subclasses weakened performance.

Discussion

There is a delicate balance to be struck here between decision speed and accuracy. Errors that are inexpensive to calculate and retract can be tolerated, but only up to a point. As the grounds for promotion and prioritization become more sophisticated (incorporating risk, utility, influence, novelty, acuity, and involvement), even greater performance improvements are expected. Caution is appropriate, however. The danger of extensive prioritization is that too many subclasses effectively degenerate a tier 3 already pruned for accuracy into a ranked list of rules. In such a case, the system may make individual decisions more quickly, but it is likely to make more mistakes and solve entire problems more slowly. For example, ACE begins with 38 heuristics in tier 3, but in some problem classes may retain as few as 15 of them during testing. The resultant structure then no longer benefits from a synergy among heuristics that work in concert, FORR's guiding principle. An example of this arises on a particularly hard problem class: 30-8-.26-.34. Here, prioritization into 8 subclasses (most of which are pairs of heuristics)

reduces solution time by about 60%. If ACE uses a ranked list of heuristics instead (effectively placing one in each subclass), the result is poorer performance.

Finally, the current version of FORR tabulates metaknowledge during learning, but applies most of it during testing. For example, heuristics are only dropped for low accuracy after learning is completed. Ideally, FORR would continually monitor and react to the metaknowledge outlined here so that it learned, discarded, promoted, and prioritized heuristics throughout the life of the application. This winnowing would apply to representations and learning methods as well. The result would be a highly-sophisticated autonomous system.

Acknowledgements

This work was supported in part by NSF IIS-0328743 and by PSC-CUNY. Thanks for their support in this work go to Gene Freuder, Tiziana Ligorio, Anton Morozov, Rick Wallace, CUNY's ACE study group, and the Cork Constraint Computation Centre, supported by Enterprise Ireland and Science Foundation Ireland.

References

Cheeseman, P., B. Kanefsky and W. M. Taylor (1991). Where the REALLY Hard Problems Are. *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence* pp. 331-337.

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

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

Epstein, S. L. (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. *Machines That Learn to Play Games*. J. Fürnkranz and M. Kubat. Huntington, NY, Nova Science: 153-178.

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

Epstein, S. L. and G. Freuder (2001). Collaborative Learning for Constraint Solving. *Principles and Practice of Constraint Programming -- CP2001*. T. Walsh. Berlin, Springer Verlag. 2239: 46-60.

Epstein, S. L., J. Gelfand and E. T. Lock (1998). "Learning Game-Specific Spatially-Oriented Heuristics." *Constraints* 3(2-3): 239-253.

Freuder, E. and A. Mackworth, Eds. (1994). *Constraint-Based Reasoning*. Cambridge, MA, MIT Press.

Keim, G. A., et al. (1999). PROVERB: The Probabilistic Cruciverbalist. *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, Orlando, AAAI Press.

Nudel, B. (1983). "Consistent-labeling problems and their algorithms: Expected-complexities and theory-based heuristics." *Artificial Intelligence* 21: 135-178.

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