Metacat: A Self-Watching Cognitive Architecture for Analogy-Making

James B. Marshall (marshall@cs.pomona.edu)
Computer Science Program
Pomona College
610 N. College Ave.
Claremont, CA 91711 USA

Abstract

This paper describes Metacat, an extension of the Copycat analogy-making program. Metacat is able to monitor its own processing, allowing it to recognize, remember, and recall patterns that occur in its "train of thought" as it makes analogies. This gives the program a high degree of flexibility and self-control. The architecture of the program is described, along with a sample run illustrating the program's behavior.

Introduction

This paper describes Metacat, an extension of the Copycat analogy-making program originally developed by Hofstadter and Mitchell (Hofstadter, 1984; Mitchell, 1993). Copycat was developed as a model of the complex interplay between bottom-up and top-down perceptual processes in the mind, which together enable humans to perceive analogies between different situations in remarkably flexible ways. The program operates in an idealized microworld of analogy problems involving short strings of letters. Although the program understands only a limited set of concepts pertaining to its letter-string world, its "fluid" processing mechanisms give it considerable flexibility in recognizing and applying these concepts in many diverse situations.

The long-term goal of the Copycat line of research is to computationally model how high-level cognitive phenomena such as creativity, analogical perception, understanding, and self-awareness can arise out of a subcognitive substrate composed of a huge number of tiny, nondeterministic processes, each of which is far too small by itself to support such phenomena. Few people would suggest that individual neurons in the brain (or individual molecules) are "conscious" in anything like the normal sense in which humans experience consciousness. One is forced to accept the fact that self-awareness arises, somehow, out of nothing but billions of molecular chemical reactions and neuronal firings. How can individually meaningless physical events in a brain—even a huge number of them—ultimately give rise to meaningful awareness? Hofstadter has argued that two key ideas are of paramount importance (Hofstadter and FARG, 1995):

> What seems to make brains conscious is *the special way they are organized*—in particular, the higher-level structures and mechanisms that come into being. I see two dimensions as being critical: (1) the fact that brains possess *concepts*, allowing complex representational structures to be built that automatically come with associative links to all sorts of prior experiences, and (2) the fact that brains can *self-monitor*, allowing a complex internal self-model to arise, allowing the system an enormous degree of self-control and open-endedness.

Work on Copycat explored the first idea through the development of a computer model of analogy-making in which the program's representation of concepts is intimately intertwined with its mechanisms for perceiving similarity between different idealized situations. Recent work has focused on the second idea by incorporating *self-watching* into the model—namely, the ability of a system to perceive and to explicitly characterize its own perceptual processes. The objective of this work has been to develop mechanisms that allow the program to monitor its own actions and to *make explicit* the ideas that come into play during the course of solving analogy problems. This can be thought of as adding a higher "cognitive" layer on top of the program's "subcognitive" layer, enabling the program to watch and remember what happens at its subcognitive level as perceptual structures are built, reconfigured, and destroyed. Humans are capable of paying attention to patterns in their own thinking in a similar fashion (see, for example, Chi *et al.*, 1994).

Self-watching in Copycat and Metacat

The Copycat architecture has been discussed at length elsewhere (Mitchell, 1993; Hofstadter and FARG, 1995), so details will be omitted here. Briefly, the program consists of a long-term memory of concepts about the letter-string world, called the *Slipnet*, together with a short-term memory for perceptual structures, called the *Workspace*. In the Workspace, small nondeterministic agents called *codelets* examine the letters of an analogy problem (*"**abc** ⇒ **abd**; **mrrjjj** ⇒ **?**"*, for example), and build

up structures around the letters representing a particular interpretation of the problem. The program's high-level behavior emerges in a bottom-up manner from the collective actions of many codelets working in parallel, in much the same way that an ant colony's high-level behavior emerges from the individual behaviors of the underlying ants, without any central executive directing the course of events.

Guiding the search for a mutually-consistent set of structures are concepts in the Slipnet, which become activated to different degrees depending on the activity in the Workspace. This activation may spread to neighboring concepts, and strongly influences codelet decisions, resulting in top-down pressure that guides the program in its search for a good interpretation of a problem.

The overall degree of Workspace organization is measured by a number called the *temperature*. Temperature not only reflects the state of the Workspace, it also continuously regulates the amount of randomness used by codelets in making decisions. At high temperatures, few Workspace structures exist, so decisions are made in a highly random manner, since not much is yet known about the problem. However, as relationships among the letters are noticed and structures are built, the temperature falls, and Copycat begins to gain "confidence" in its understanding of the situation. At lower temperatures, decisions are still probabilistic, but are much less random, being strongly biased by the estimated promise of newly emerging structures, all of which compete for attention by codelets. At very low temperatures, codelets pay attention to only the most promising structures, and decisions become largely deterministic. Thus the type of strategy used by the program to explore its search space ranges along a broad continuum, from being very diffuse and highly parallel at high temperatures to being very serial and focused at low temperatures.

To summarize, Copycat's search proceeds via a large number of fine-grained stochastic decisions, which depend on the temperature. These decisions may cause new structures to be built or existing structures to be destroyed, which changes the temperature and subsequently alters the course of structure building, forming a kind of feedback loop. Temperature thus serves as a very crude mechanism for self-watching in Copycat, since it allows the program to regulate its own behavior to a limited extent. That is, by tying the stochastic activity of codelets to the temperature, the program becomes sensitive to the consequences of its own actions, since the temperature reflects the result of these actions, albeit in a very coarse way (*i.e.*, in the form of a single number).

This type of rudimentary self-watching, however, is quite primitive. Copycat can characterize patterns within its perceptual input (the letter strings), but is completely oblivious to patterns that arise in its *processing* of that input. For example, when solving the problem *"**abc**⇒ **abd**; **xyz**⇒ **?**"*, Copycat usually attempts to take the successor of ***z***, which is impossible in the program's microworld. It "hits a snag", and is forced to try something else. However, it often just tries the same thing again, over and over, sometimes as many as thirty or forty times before stumbling by chance on an alternative approach (such as the answer ***xyd***). Unlike humans, the program is unable to recognize when it has fallen into a repetitive pattern of behavior. It has no memory of its actions over time, and thus cannot recognize when it has encountered the same situation in the past. As a result, Copycat lacks insight into how it arrives at its answers, and consequently cannot explain what makes one answer better or worse than another.

In contrast, Metacat is able to create much richer representations of the analogies it makes, enabling it to compare and contrast answers in an insightful way. This has involved incorporating an episodic memory into the original Copycat architecture, along with new mechanisms that allow the program to monitor itself, so that it can recognize, remember, and recall patterns that occur in its "train of thought" as it makes analogies.

To do this, Metacat creates an explicit sequential record of the most important processing events that occur during a run. This temporal record is examined by codelets for patterns—in the same way that Copycat's codelets examine letter-strings for patterns—and serves as the basis for constructing an abstract description of an answer in terms of the key concepts and events that led to its discovery. Furthermore, by monitoring its own processing in this way, Metacat can recognize when it has become "stuck in a rut", enabling the program to break out of the rut by focusing on ideas other than the ones that seem to be leading it nowhere.

The Architecture of Metacat

Metacat's architecture includes all of Copycat's architectural components, such as the Workspace and the Slipnet, as well as three new components: the *Episodic Memory*, the *Themespace*, and the *Temporal Trace*. When the program discovers a new answer, it pauses to display the answer along with the Workspace structures that gave rise to it. These structures represent a way of interpreting the problem that yields the answer just found. All of this information is then packaged together into an *answer description* and stored in the Episodic Memory, after which the program continues searching for alternative answers to the problem, instead of simply quitting. Gradually, over time, a series of answer descriptions accumulates in memory, each one containing much more information than just the answer string itself.

The most important structures stored in answer

descriptions are called *themes*, which represent the essential ideas underlying an answer. The collection of themes associated with an answer serves as the basis for comparing it to other answers stored in memory. Furthermore, Metacat may be reminded of other answers it has encountered in the past if the themes associated with a newly discovered answer, acting as a memory retrieval cue, are similar enough to those of a previously stored answer description. Thus an answer's themes act as an index under which it is stored and retrieved from memory.

Themes get created in Metacat's Themespace as codelets build structures around the letter-strings, and are composed of Slipnet concepts. For example, in the problem *"**abc**⇒ **abd**; **xyz**⇒ **?**"*, an *Alphabetic-Position: opposite* theme representing the idea of alphabetic-position symmetry between the letters ***a*** and ***z*** might get built if the program perceives these letters as playing analogous roles in their respective strings (an interpretation that may lead to the "mirror image" answer ***wyz***).

In some ways, themes are like ordinary Workspace structures. They are not initially present in the Themespace; rather, they arise during the course of a run as the result of codelet activity occurring in the Workspace. In other ways, however, themes behave like Slipnet concepts. They can take on different levels of activation, reflecting the extent to which the ideas they represent are supported by structures in the Workspace. A theme's activation level decays over time, and is influenced by the activation levels of other themes. Like Slipnet concepts, themes can, under certain conditions, exert strong top-down pressure on perceptual activity in the Workspace. In fact, themes can assume both positive and negative levels of activation, ranging from −100 to +100. A positively-activated theme exerts "positive thematic pressure", encouraging the creation of Workspace structures that support the idea represented by the theme. A negatively-activated theme, on the other hand, exerts "negative thematic pressure", which *discourages* the creation of structures related to the theme, promoting instead the creation of alternative structures.

The Temporal Trace serves as the focal point for self-watching in Metacat. Like the Themespace, the Trace accumulates information over the course of a single run, and can be viewed as an extension of the Workspace. The Trace stores an explicit temporal record of the most important processing events that occur while the program works on an analogy problem. Examples of such events include recognizing some key idea pertaining to the problem (by noticing the strong activation of a theme or concept, for instance), hitting a snag, or discovering a new answer. Once processing events have been explicitly represented in the Trace as "reified" structures in their own right, they are subject to examination by codelets as well. Metacat thus uses a single set of mechanisms for perceiving patterns in its perceptual input and in its own processing of that input. When a new answer is found, an answer description can be formed by examining the temporal record in the Trace to see which events contributed to the answer's discovery.

This approach is similar in flavor to work on derivational analogy, in which the trace of a problem-solving session is stored in memory for future reference, together with a series of annotations describing the conditions under which each step in the solution was taken (Carbonell, 1986; Veloso, 1993). In Metacat's case, however, the information in the Trace is used as the basis for constructing an abstract description of the answer found, rather than being permanently stored itself.

One way to appreciate the abstract, chunked nature of the information in the Trace is to consider the number of "steps" that occur during a typical run of Metacat. At a very fine-grained level of description, where each step corresponds to an action performed by a single codelet, a run consists of many hundreds or thousands of steps. At this level of description, no two runs are ever *exactly* the same, even if they involve the same letter-strings (unless, of course, both runs start with the same random number seed). On the other hand, at the level of description of the Trace, a typical run consists of a few *dozen* steps. At this level of granularity, each step corresponds to a single event represented in the Trace—each of which arises from the actions of many codelets.

For example, Figure 1 shows the contents of the Trace after a run on the problem *"**abc**⇒ **abd**; **xyz**⇒ **?**"*, in which the program, after trying unsuccessfully a couple of times to take the successor of ***z***, answers ***xyd***. The events that occur during the run appear left to right in chronological order. Although this run involves a total of 1,558 codelets, the high-level picture shown in the Trace consists of just twelve events, which represent the "major milestones" encountered along the way in the program's search for an answer. Such events include the activation of concepts in the Slipnet, perceiving entire strings as single, chunked wholes, creating new rules for describing string changes, hitting a snag, and discovering a new answer.

For instance, as can be seen in the figure, the Slipnet concept *identity* gets activated early on in this particular run (due to the program perceiving the ***a***'s and ***b***'s in ***abc*** and ***abd*** as corresponding). This is followed by the perception of ***abc*** and ***xyz*** as *predecessor* groups going in the same direction (to the left). The next event records the creation of the rule *Change letter-category of rightmost letter to successor* for describing ***abc*** ⇒ ***abd***, which leads inevitably to a snag. In the aftermath of the snag, another rule is created (*Change letter-category of rightmost letter to 'd'*), and ***abc*** and ***xyz*** are reperceived as *successor* groups (again going in the same direction—only

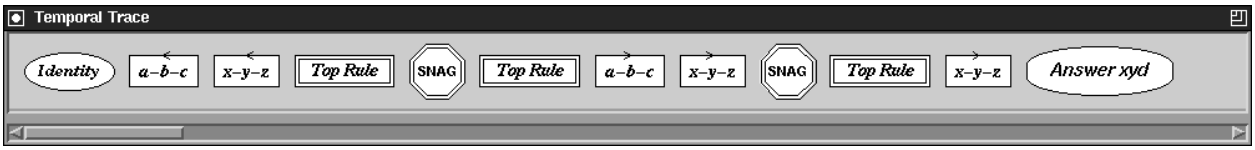

Figure 1: The temporal record of a run on the problem *"**abc*⇒ *abd; xyz*⇒ *?"*.

this time to the right). However, the program again attempts to use the first rule, resulting in another snag. Finally, after creating yet another rule and again perceiving ***xyz*** as a successor group, the program finds the answer ***xyd***.

Pattern-clamping and Self-control

The Trace allows Metacat to monitor the subcognitive processing activity in the Workspace at a very abstract and highly-chunked level of description, enabling the program to "see" what it is doing during a run. Equally important, however, is the program's ability to *respond* to what it sees by clamping particular themes and concepts at high activation, resulting in strong top-down pressure on processing. Various types of *patterns* consisting of sets of themes, concepts, or codelet types can be clamped by the program in response to different situations that arise. Clamping a pattern alters the probabilities that certain types of codelets will run, or that certain types of Workspace structures will get built, effectively steering the behavior of the program in particular directions. This may lead the program to revise its interpretation of a problem, by catalyzing the reorganization of structures in the Workspace in accordance with the ideas represented by the pattern.

Metacat's ability to revise its perception of a situation in response to events in the Trace affords the program a very powerful degree of self-control. Patterns—especially patterns of themes—act as a "medium" through which the program is able to wield control over its own behavior. For example, in the problem *"**abc*⇒ *abd; xyz*⇒ *?"*, the program usually perceives ***abc*** and ***xyz*** as going in the same direction at first, which leads to a snag (as in the run shown in Figure 1). This interpretation of the problem, based on the idea that letters having identical positions in their respective strings correspond to one another (***a*** to ***x***, ***b*** to ***y***, ***c*** to ***z***), is characterized by a *String-Position: identity* theme. When an event is recorded in the Trace, the themes most active at the time of the event are also noted along with it. These themes serve as the event's *thematic characterization*. In the case of a snag event, the thematic characterization represents an interpretation of the problem that has just led to failure. If Metacat continues to hit the same snag several times in succession, a series of snag events will accumulate in the Trace, all with very similar thematic characterizations. This similarity may be noticed by codelets (the probability becoming higher as more snags accumulate), causing them to take action by clamping the "offending" themes (such as *String-Position: identity*) with strong negative activation. This encourages the program to explore alternative interpretations of the problem by steering it away from the ideas causing the snag, which may subsequently lead it to the discovery of other answers, such as ***wyz***. In this way, Metacat can recognize its own repetitive behavior and respond accordingly.

Two types of codelets are responsible for examining and responding to events unfolding in the Trace. The first type, called a *Progress-watcher*, is responsible for deciding whether or not to unclamp a clamped pattern. If a *Progress-watcher* codelet runs while a pattern is clamped, it examines the most recent event in the Trace to determine how much time has elapsed since the event occurred. Generally speaking, the purpose of clamping a pattern is to precipitate a series of events that reorganize the perceptual configuration of the Workspace in some way. It is therefore better to wait until the structure-building activity occurring in the wake of a clamp has settled down before concluding that the clamp has "run its course". Accordingly, if the amount of time since the most recent event in the Trace is less than some minimal settling period, then the codelet simply fizzles, leaving the clamped pattern still in effect. On the other hand, if enough time has passed without any new important events having transpired, the codelet unclamps the pattern and then determines the amount of progress that was made since the clamp occurred. Depending on the amount of progress achieved, the codelet may decide to post a follow-up codelet in order to see whether a new answer can be made based on the newly-created structures.

The criteria for judging the success of a clamp can vary. Sometimes, the purpose of clamping a pattern is to promote the creation of *specific* types of Workspace structures. Other times, the purpose is to encourage the creation of structures of *any* type, so long as they are compatible with the clamped pattern. The progress achieved by a clamp can be measured by observing the number of structures that get built in the immediate aftermath of the clamp, and the extent to which they are compatible with the pattern.

If no patterns are clamped when a *Progress-watcher* codelet runs, then instead of checking on the progression of events in the Trace, the codelet

checks on the current rate of structure-building activity in the Workspace. This activity is measured by a single number that serves as a quick estimate of the "freshness" of the current Workspace structure configuration. More precisely, it is an inverse function of the average age of the most recently created structures. Thus the activity level tends to remain high as long as new structures are being built, but eventually drops to zero in the absence of new structures.

If the activity level is zero, indicating that nothing much is happening in the Workspace, then Metacat may have arrived at an impasse in its search for answers to the current problem. This is not quite as bad as hitting a snag, but it still ought to prod the program into trying something different. However, in the case of an impasse, there is usually no clear set of "offending" structures or themes on which to pin the blame, unlike in the case of a snag. Indeed, the impasse may well arise from a *lack* of appropriate structures, rather than from the existence of the "wrong" structures.

Therefore, in the absence of Workspace activity, *Progress-watcher* codelets check to see whether particular types of new structures are needed. For example, a codelet may examine the quality of the rules that have been built so far. If no good rules yet exist, the codelet may try to encourage the creation of better rules by clamping a pattern that strongly increases the probability that rule-seeking codelets will run, while simultaneously inhibiting other types of codelets. Eventually, other *Progress-watcher* codelets will turn off the clamp once enough time has passed without any more events being added to the Trace. Since this type of clamp is only concerned with the creation of new rules, the amount of progress achieved is judged solely on the basis of the quality of the rules that get created in the clamp's wake.

The second type of codelet that "watches the action" from the high-level vantage point of the Trace is called a *Jootser* (short for "*j*umping *o*ut *o*f *t*he *s*ystem"). These codelets are responsible for noticing repetitive behavior that the program has fallen into. An example of such behavior arising from a snag was sketched above. However, *Jootser* codelets are sensitive to other kinds of situations as well. For example, it is possible for Metacat to become "fixated" on some idea, such that it ends up clamping the same pattern over and over again, without making any significant progress. In this case, too, *Jootser* codelets may notice the series of recurring events in the Trace and take action.

For instance, if an analogy problem happens to involve a string that changes in some difficult-to-describe way, the program may end up repeatedly clamping patterns in an attempt to spur the creation of better rules for describing the change. Repetitive clamping behavior can even arise from unsuccessful attempts to break out of a cycle of snags. That is, clamping a pattern in response to a recurring snag may prove to be ineffective, leading only to further snags and more pattern-clamping, rather than to a new interpretation of the problem. Faced with several similar clamp events in the Trace, a *Jootser* codelet decides probabilistically whether to "joots" based on the number of clamps and the average amount of progress achieved by each. The more clamp events there are, the more likely jootsing is to occur, especially if the amount of progress is low, unless recent clamps appear to be making more headway than earlier ones. Unlike jootsing from snags, however, jootsing from a series of recurring clamp events does not involve the clamping of any new patterns in response. Instead, Metacat simply "gives up" in a graceful manner and stops.

In a sense, Metacat's ability to respond to a recurring snag by focusing on alternative ideas can be thought of as "first-order" jootsing. In contrast, the program's ability to eventually give up when it recognizes that its repeated attempts to circumvent a snag are leading nowhere can be thought of as "higher-order" or "meta-level" jootsing (*i.e.*, jootsing from repeated unsuccessful jootsing). The important point is that the same general mechanisms are responsible for first-order and meta-level jootsing in Metacat—namely, *Jootser* codelets and the explicit representation of processing events in the Trace.

An Example of Jootsing

The following example illustrates the idea of jootsing. In this run, Metacat is given the problem *"**eqe** ⇒ **qeq**; **abbbc** ⇒ **?**"*. The program builds up an interpretation of the string ***abbbc*** as a successor group composed of the letter ***a***, the group ***bbb***, and the letter ***c***. The two rules shown below are also created to describe the ***eqe*** ⇒ ***qeq*** change:

- *Swap letter-categories of all objects in string*

- *Change letter-category of leftmost letter to 'q'*
  Change letter-category of middle letter to 'e'
  Change letter-category of rightmost letter to 'q'

Around time step 1100, the program attempts to apply the first rule to ***abbbc***, which results in a snag, since the idea of a three-way swap involving the letters ***a***, ***b***, and ***c*** makes no sense (see Figure 2). Of course, if it had chosen to use the second rule instead of the first, then it would have found the answer ***qeeeq***, but it prefers the first rule, since this rule is more abstract.

Over the next 3000 time steps, the program tries again and again to swap the letters of ***abbbc***, often breaking various structures in the process, but always rebuilding them in the same way as before. Eventually, at time step 4280, a *Jootser*

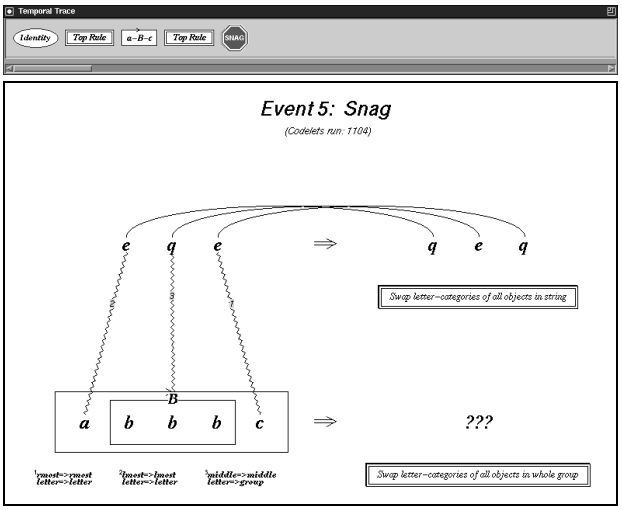

Figure 2: Attempting to swap the letters of ***abbbc***

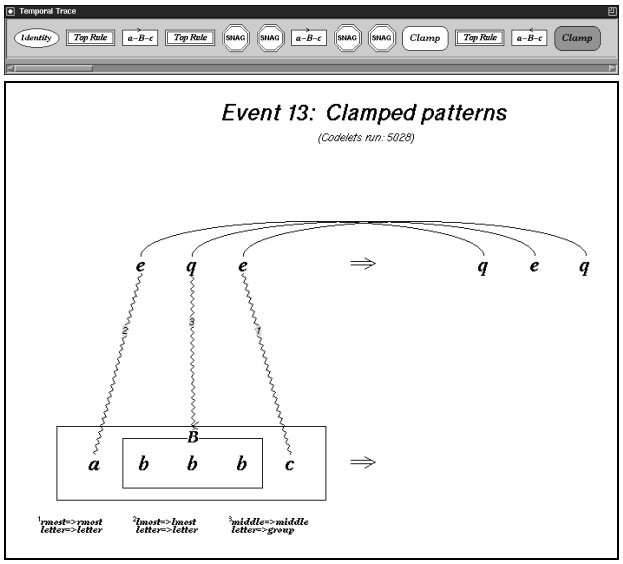

Figure 3: Clamping patterns in response to snags

codelet notices the pattern of recurring snag events in the Trace, all of which involve the themes *String-Position: identity*, *Object-Type: identity*, and *Object-Type: different*. These themes arise from the program's interpretation of the letters ***e***, ***q***, and ***e*** in ***eqe*** as corresponding, respectively, to the letter ***a***, the group ***bbb***, and the letter ***c*** in ***abbbc***. The *Object-Type: identity* theme is based on the ***e***−***a*** and ***e***−***c*** correspondences, while the *Object-Type: different* theme results from the correspondence between ***q*** and ***bbb***, since one is a letter and the other a group.

In hopes of finding a way around the recurring snag, the codelet decides to negatively clamp the *Object-Type: identity* theme. In the wake of the clamp, ***abbbc*** is reinterpreted as a predecessor group going to the left, and a new rule is created to describe ***eqe*** ⇒ ***qeq***, but these new structures do not really change the basic situation. Soon afterwards, another *Jootser* codelet tries again, this time clamping *both* *Object-Type* themes, which essentially "paralyzes" the program for the duration of the clamp, since no structures can be built that are compatible with both of these themes simultaneously. Figure 3 shows the state of the Workspace and Trace at the time of the second clamp.

A few hundred codelets later, the program hits the snag again. This is followed shortly thereafter by another clamp. This clamp, like the one before it, achieves no new progress. After hitting the snag yet again, the program finally decides to give up. More precisely, at time step 5933, a *Jootser* codelet notices the three clamp events in the Trace, all of which involve overlapping thematic characterizations. Moreover, neither of the two most recent clamps have resulted in any discernible progress, which further increases the probability of jootsing. Consequently, the program prints the message "this is getting boring, I can't think of anything else to try" and then ends the run.

As this example shows, Metacat is able to realize when it is "stumped", instead of just cycling endlessly. The program's ability to monitor its own processing at an abstract level of description affords it a great deal of flexibility and self-control, and, it is to be hoped, represents a step toward the goal of understanding the cognitive mechanisms underlying human self-awareness.

References

Carbonell, J. (1986). Derivational analogy: a theory of reconstructive problem solving and expertise acquisition. In R. Michalski, J. Carbonell, & T. Mitchell (Eds.), *Machine learning, volume 2*. San Francisco: Morgan Kaufmann.

Chi, M., de Leeuw, N., Chiu, M.-H., & LaVancher, C. (1994). Eliciting self-explanations improves understanding. *Cognitive Science*, 18:439–477.

Hofstadter, D. R. (1984). *The Copycat project: an experiment in nondeterminism and creative analogies*. AI Memo 755, MIT Artificial Intelligence Laboratory.

Hofstadter, D. R. & the Fluid Analogies Research Group (1995). *Fluid concepts and creative analogies*. New York: Basic Books.

Marshall, J. (1999). *Metacat: a self-watching cognitive architecture for analogy-making and high-level perception*. Doctoral dissertation, Department of Computer Science, Indiana University, Bloomington. http://www.cs.pomona.edu/marshall/metacat.pdf

Mitchell, M. (1993). *Analogy-making as perception*. Cambridge, MA: MIT Press/Bradford Books.

Veloso, M. (1994). *Planning and learning by analogical reasoning*. Berlin: Springer-Verlag.