# Learning by analogy – making Copycat curious

**Greg Detre**
Harvard University, Cambridge, MA
email: greg@gregdetre.co.uk

**Abstract**: The aim of this paper is to consider a future implementation of a curious machine that learns through analogy-making. I will describe what I mean by curiosity, and consider how the state of the art in computational analogy-making provides a good starting-point for the endeavour by considering how one particular such architecture, Copycat, might be extended. I will focus on how a future system might begin to self-organise and learn, reducing the reliance on human hand-coding of parameters, and making steps towards becoming genuinely domain-general, and how such a system could be considered to be being curious.

**Keywords**: curiosity, analogy-making, learning, concepts, case-based reasoning, derivational analogies

*A year spent in artificial intelligence is enough to make one believe in God – Anon.*

## Introduction

This paper was originally written for a seminar class at the MIT Media Lab, entitled ‘Curious machines’. The aim was to explore what curiosity is, its role in intelligence and how curiosity might be implemented in future AI systems, asking questions like ‘How can we build machines that are as curious learners as natural systems? How can we build systems that have a deeper understanding of the learning process beyond turning the statistical crank of a learning algorithm?’ – see http://courses.media.mit.edu/2003spring/mas963.

I’m going to start by outlining some of the features that a recent discussion highlighted as being characteristics of curious behaviour, before proposing a pithy and restricted characterisation of what it is to be curious that will guide the discussion in the rest of the paper. The intention is to discuss what would be necessary to make Hofstadter and Mitchell’s Copycat analogy-making model be a curious learner.

### Curiosity

Being curious is about seeking knowledge that you don’t even know for sure that you’ll ever need. Curiosity in its fullest sense presumably requires a learning system and cognitive architecture complex enough to subserve goal-directed and flexible behaviour, recognition of novelty, and some degree of (self-)evaluation. Being curious is proactive and explorative, rather than a reaction to immediate need. The more complex and diverse the goals, behaviours and representations, the more complex the curiosity manifested. There can be different types of curiosity, triggered under different circumstances, suited to different domains, goals or learning styles. It also seems intuitively unlikely that curiosity has a unitary substrate, but this is possibly contentious and tangential to the points that I want to make next, so I won’t discuss it.

In its barest form, a curious machine is one that *interrogates its environment*. ‘Interrogation’ has the requisite sense of a directed enquiry about things that you want to know from something/someone that has the answers, out of which an internal understanding is built. Interrogations are about things that you don't know about now, and don't necessarily need to know, but may well come in useful in the grand scheme of things. There's also a sense of a dialogue, in which you ask questions, narrow down the domain of enquiry, realise where you’re ignorant, ask more questions about the new things you don't understand, and accommodate this new knowledge.

I hope that this will prove an illustrative rather than misleading way of starting to think about the business of being curious. It prompts certain observations. Importantly, you need to know some things already in order to know what to ask and to make sense of the answers. Indeed, what you know already will make a big difference to the kinds of questions it occurs to you to frame, and the way you interpret the answers. The order in which you are told things can drastically affect how easy it is to draw conclusions from them. It helps you and your interlocutor gauge your understanding to be tested. If you ask the right questions, you can draw conclusions that may have general applicability very quickly. A popular way of answering a question is to provide examples and counter-examples. Examples may make you see things in a new light, and counter-examples help you map out the boundaries within which new knowledge applies. Figuring out which questions to ask, especially figuring out when something crucial is missing from your understanding, is hard but important. For this reason, an unexpected answer can be very instructive. If you find it difficult to understand something, good teachers will explain it to you by saying it in another way, or showing how it's similar (or different) to something you already understand, or composed out of other things. You need to know when to stop asking questions as well as when more questions are needed. I have framed this idea of a question-and-answer interrogation in a very linguistic light, but it needn't be at all.

## Analogy-making

There are lots of ways to think about things and to learn to deal with new problems and concepts. Minsky [forthcoming, chapter 7] details a number of them, of which I've reproduced a modified sample:

breaking it down into smaller parts

solving a simplified version, then generalising, or just dealing with the extra complexities and exceptions one by one

seeing how it's similar to something you already know about, or reformulating the problem in a different domain

describing it in a more abstract way (e.g. formal logic)

meta-reflection – considering what makes a problem hard and where you're going wrong

considering whether the problem is really worth solving at all

extensively searching through possibilities

imagining how someone you respect would tackle the problem

If we could build a system that was able to do all of these things flexibly and in different domains, we'd be well on our way towards a very healthy IPO. My focus here will be on the top two-thirds of this list, which I consider to fall loosely under the general idea of 'learning by analogy'. This very general suite of approaches has been termed computational analogy-making [French], case-based reasoning [Leake], remembering and adapting [Kolodner], high-level perception [Chalmers, French and Hofstadter], and structure mapping theory [Gentner], amongst other names.

I consider learning by analogy to be amongst the most central and powerful representational and learning tactics we employ. In a perverse way, this is evidenced by the effortlessness and invisibility of the processes that see these similarities and analogies. We constantly think of concepts in terms of other concepts, ignoring what is irrelevant to the comparison, subconsciously but effortlessly alighting on what is salient. We jump up and down levels, into different modalities and across mental realms. Human language's concise expressiveness rests in part on words being reused for new purposes, enriching their associations and coopting their 'inferential machinery' [Minsky, Jokes and the cognitive unconscious]. This has been noted before, most eloquently by Hofstadter [1979, 1995; also quoted in Marshall, 1999].

### Higher-level perception and Structure Mapping Theory

I intend to take Hofstadter and the FARG's implementations as my starting point for discussion of a future implementation of a curious machine that learns through analogy-making, but before going any further, we need to briefly survey the main debate in the computational analogy-making literature, which is best characterised by a comparison between the 'Structure Mapping Theory' and 'Higher Level Perception' camps [French, 2002].

According to SMT, an analogy is an 'alignment of relational structure' [Gentner & Markman, 1997]. Here, the relations are the internal links that determine the composition and arrangement of the structure, which are contrasted with the 'attributes' and 'object descriptions' which determine 'mere-appearance matches'. Morrison & Dietrich [1995] consider that Gentner et al.'s aim is to present a model of the *comprehension* (rather than the discovery) of analogy, where for a given structure, the system is able to retrieve a stored match for which the mapping of relations is closest. Their implementation, SME, starts by seeing many local matches out of which a consistent large-scale structure coalesces, and appears to mirror certain salient experimental results with human subjects.

In contrast, Hofstadter and the FARG [Hofstadter, 1995] want to cast analogy-making as playing a much more central and less specialised role – "analogy-making is going on constantly in the background of the mind, helping to shape our perceptions of everyday situations. In our view, *analogy is not separate from perception: analogy-making itself is a perceptual process*" [Chalmers, French and Hofstadter, 1991].

This needs a little explaining. The central point is that the process of building up a compound representation of a situation or scenario cannot be independent of the process of seeing a mapping between scenarios. Both of these processes are intertwined as 'high-level perception'. High-level perception begins at that level of processing where concepts begin to play an important role. This is pretty nebulous, but that's fine. We can see concepts as being abstract or concrete, simple or complex – any aggregation, processing or filtering of raw sensory data can be seen as conceptualising.

The two problems in high-level perception are the problems of relevance and of organisation:

1. *relevance* – how do you determine what's salient within the morass of low-level data, and pick it out to pass on to higher levels of processing?
2. *organisation* – how do you organise all of that (multi-modal) data together, i.e. how do you determine what to clump together and what's related to what?

These two problems are critical for SMT, since if the wrong aspects of the perceptual data are chosen, or if they are organised poorly, no analogies will ever be found. Yet this is out of SME's control, because it artificially separates the processes of human hand-coding of perception and its own mapping. The same criticism Hofstadter made of Bacon [Langley et al., 1987] could be made of SME, namely that it "was fed precisely the data required to derive the [Kepler's] law" [Hofstadter, 1995].

In other words, analogy-making requires representations to be built dynamically, extracting, organising and *reorganising* what's salient about the current situation based on the current context, goals, beliefs, and at the same time as trying to perform tentative mappings with past situations and knowledge. To *comprehend* an analogy is to *discover* it – you can't do the former in any rich, flexible or meaningful way without doing the latter.

## Copycat

Copycat is intended to illustrate how the various strata of such a view of analogy-making as high-level perception could operate and interact, involving:

- the gradual building-up of representations
- the role of top-down and contextual influences
- the integration of perception and mapping
- the exploration of many possible paths towards a representation
- the radical restructuring of perceptions, when necessary

Copycat considers analogies like the following:

*abc : abd :: ijkk : ?*

Most people would prefer *ijll*, but would recognise the validity of *ijkl*, *ijkd*, *ijdd* or *abd*, to name just a few. Copycat's architecture is designed to allow top-down and bottom-up influences to interact, constraining a search[1] through the space of possible mappings between letter-strings, and so producing a mapping to a new string, as well as providing a rating of the system's 'happiness' with its solution. This could be seen in three main (concurrent) tasks:

1. build a representations of the three starting strings
2. describe how to map from the source to target strings
3. apply the same transformation to the third string

There are a number of things about the Copycat architecture that are special or interesting. It's split into three parts:

### *Slipnet*

This is the high-level, long-term conceptual memory of Copycat (*see Fig. 1*), represented as a semantic network. It contains concepts like 'successorship', 'rightmost', 'opposition' and 'symmetry', each of which are linked together by proximity (i.e. association) weights. Each concept has a pre-assigned 'conceptual depth' and activation. The conceptual depth is a sort of aesthetic, subjective, hand-coded value intended to capture how abstract or interesting a concept is. The activation reflects the extent to which the concept appears to be relevant to the current problem, and how activated nearby/associated concepts are.

### *Coderack*

The Coderack is the repository for the codelets – these are small, specific pieces of code that carry out low-level tasks. Some codelets look for particular patterns, or evidence that a given concept may be playing a role somewhere, while others build bonds and groups within a string, or bridges/correspondences between strings, and finally some break these structures back down again when Copycat seems to be hitting an impasse.

Each codelet is selected probabilistically from the Coderack according to its 'urgency', which is partly hand-coded, and partly a function of the current activations and deformations in the Slipnet, and partly affected by the preceding codelets which triggered it.

### *Workspace*

This is a sort of scratchpad on which the codelets operate, containing the strings, and the structures built up between them. The strength of a structure is a function of the activation and conceptual depth of the related concept (e.g. sameness, successorship), how long it has lasted, whether it conflicts with other structures, amongst other factors. Structures can be nested. I find it useful to think of the Workspace structures as tokens of Slipnet concept-types.

[1] Although Hofstadter avoids the word 'search' in the context of thinking because of the connotations of formal, efficient techniques for searching well-defined spaces, that he rejects [Kelly, 1995].

Copycat is great at interacting top-down and bottom-up, being mostly sensible but not myopically systematic, and building structures so that they 'flex' in the right places. Hofstadter terms the system's overall approach a *parallel terraced scan*, which can be understood in search terms as exploring the most promising avenues proportionally/probabilistically more. Where the agenda of a depth-first search is a stack, and breadth-first uses a queue to decide the next node, the parallel terraced scan uses a stochastic priority-queue of codelets, ordered by their 'urgency'. These priorities are based on the bi-directional interactions between the top-down associations and concept activity-values in the Slipnet and the happiness and salience of the bottom-up structures built by the codelets.

Finally, the *temperature* is a measure of the richness and internal coherence of the structures that have been built up so far in the Workspace. When these structures are weak, employing conceptually shallow concepts, and when large parts of the strings haven't been accounted for or don't fit, the temperature is high, making all the processes more stochastic, and increasing the urgency of dismantler codelets. As the system builds more coherent structures, the temperature drops, and the decisions become more deterministic and less destructive. The temperature can then be seen as a kind of measure of the system's happiness with the solution it has found. As a result, Copycat may find a less satisfying analogy quite often (it has no memory of past solutions), but occasionally stumble across a highly satisfying solution, mirroring results with human experimental subjects.

An example should suffice to convey the difference between more common and more satisfying solutions. If faced with the problem:

*abc : abd :: xyz : ?*

most people's first choice would probably be *xya*, since we want to find a successor to the rightmost letter and so we loop back through the alphabet. However, a circular link from 'z' to 'a' has been deliberately excluded from Copycat's conceptual model, which forces people to think harder.

As a result, Copycat frequently builds up a set of structures on the Workspace that lead it to seek the successor of the rightmost letter, only to hit an 'impasse' (*see Fig. 2*). This happens often because Copycat's parameters are set so that it sees successorship groups more readily than predecessorship groups, which is intended to reflect human (especially Western) preferences for incrementing over decrementing and left-to-right over right-to-left.

As a result, the solutions it comes up most commonly include *xyz*, *xyy*, *xyd* and *abd*. However, there is a solution that many people find very satisfying once they see it, though few people notice it immediately, which is *wyz*. This requires a mini paradigm shift. The impasse occurs because *abc* is described as a group of successors heading rightwards from the first letter of the alphabet, and the most obvious mapping is to see *xyz* correspondingly as a group of successors heading rightwards ending on the last letter of the alphabet. In order to scale the impasse, *xyz* has to be reconceptualised as a group of predecessors heading leftwards from the last letter of the alphabet. This is exactly symmetrical to the description of *abc*, prompting a reversal of the rule from *abc* to *abd* of 'replace the rightmost letter with its successor' to 'replace the leftmost letter with its predecessor'. When I first saw this, I certainly felt that the choice of the seemingly uninteresting letterstrings microdomain as allowing for complex, psychologically plausible constructions was vindicated. Copycat finds the less satisfying solutions more often, but when it does find the *wyz* solution its satisfaction with the solution (as measured by a lower temperature) is much higher [See Hofstadter, 1995; and Mitchell, 1993 for a plethora of further letterstring puzzles that Copycat can solve].

## Metacat

Metacat [Marshall, 1999] is the second generation of Copycat, differing in a couple of important respects.

Firstly, Metacat is able to produce multiple answers for a given problem in a single run, reporting each and carrying on. In contrast, Copycat would stop each run every time it found an answer, starting each run afresh and blissfully ignorant of past successes and failures.

Secondly and crucially, Metacat builds a 'trace' of its operations as it goes along, capturing both an abstraction of the process of discovery as well as the nitty-gritty details of the state of the whole Copycat system. These extra levels of self-watching and remembering have a number of advantages. Metacat is able to avoid getting trapped in a loop or freezing when faced by an impasse that it has encountered before. In contrast, when Copycat tries to find the successor of 'z' and fails, the temperature slowly rises, certain structures that led to this dead end become more likely to be dismantled, and more often than not it retraces some of its steps only to try the very same tactic in a few iterations' time. Furthermore, by maintaining a trace of its activity as well as details of past runs, Metacat is able to use past experience to avoid this folly, and head towards a known solution or try new avenues. The most important concepts employed in a given solution are termed 'themes' – by storing the themes along with the salient events and steps in a given run, Metacat is trying to capture the essential features of a given situation, allowing limited comparison between different solutions to the same problem.

### Magnificat

Very recently, I discovered an essay online discussing vague plans for the latest implementation in the Copycat family, named Magnificat [Roberts, unpublished]. Pleasingly, many of the aims and high-level ideas raised dovetail with those discussed here, though it also contains a number of architectural innovations that deserve greater consideration than I have room for. I will consider them in passing if I feel they can especially help.

### Lessons from the Copycat family

I consider Hofstadter's work on analogy-making to be the richest source of ideas and the most impressive implementation around at the moment. Although he focuses on micro-domains, and makes no claims to have modelled the thought processes of great scientists from the past [cf Langley, 1987], there is a sense that the kinds of analogies that Copycat is able to see involve a directed and really quite human-like exploration of a far greater space.

One of the most important lessons I want to take away from Hofstadter et al.'s work is the need to build up a representation of the situation, with both top-down and bottom-up influencing each other, in order to be able to choose what's relevant for the current analogy. The flipside of this is that concepts become rich and meaningful by virtue of the way in which they can be decomposed and combined into, and influence other concepts. I'm also conscious of the adaptability of Copycat's basic architecture – at least in theory, with different concepts, codelets and their associated parameters, Copycat could be adapted to more or less any domain. Furthermore, Hofstadter claims that the parallel terraced scan is fairly resistant to problems of combinatorial explosion.

### Limitations of the Copycat family

However, even within the letterstrings domain, Copycat/Metacat is limited. There are concepts that we might expect it to have that it lacks. For instance, Copycat can't deal with sequences that aren't successors or predecessors, e.g. sequences that skip every other letter. Importantly, it can't deal with mappings involving more than one letter change, though apparently Metacat improves upon this. It can't deal with certain types of noise, nor represent interleaved sequences, e.g. *abacadae*. It can't devise analogies of its own, nor learn from counter-examples. Finally, Metacat's ability to see compare analogies is limited, as is its ability to search for past cases.

Some of these issues are quibbles, but some mask deep-seated limitations of the architecture. After all, Hofstadter explicitly states that he is not especially considering the issues of learning or self-organisation. Unfortunately, the Copycat system relies heavily on a large number of hand-coded parameter values that determine the various probabilities and relations between the system's different processes and the structures they build. The system's impressively human-like performance on a number of abstract, difficult problems very rich in internal structure (despite the limited domain) depends upon these preset, subjective, tweaked values, as well as a number of implicit judicial decisions with regard to the kinds of concepts that such a system should look for. Indeed, the choice of concepts and codelets was guided by five difficult sample problems (including the *xyz* one detailed above) that they wanted Copycat to be able to solve. Porting Copycat to a different domain, or expanding the letterstrings domain, while maintaining the delicate balance between the current concepts, would always be a labyrinthine labour-intensive task.

## Building on Copycat

I'm not going to try and tackle all of these limitations. The goal of the discussion in this paper is to consider how a curious, learning analogy-making system might be incorporated as a component in a much larger and more domain-general learning system. Unfortunately though, the problem of getting Copycat to learn or self-organise to adapt itself to new domains is a deep one. I'm going to identify a number of different levels at which some sort of learning or self-organising mechanism (or set of mechanisms) would be necessary, and then discuss how easy it would be to build a system that could learn to operate in different and potentially more complex domains than the letterstrings based on some of these ideas.

In the grand tradition of the Fluid Analogies Research Group of giving their projects capricious and unnecessarily clever names, I will refer to this sketch of a curious, self-organising, domain-general system that learns by making analogies as 'CuriousCat'.

### Searching through the parameter space

If we were to take the results from even a small sample of human subjects on a suite of letterstring problems, we could use reinforcement learning [Sutton and Barto, 2002] or genetic algorithms (GA) [Holland 1975; Koza 1997] to search the space of parameters to find the combinations that match up with the experimental data. Indeed, Hofstadter et al. appear to have the experimental data to do this. In their discussion of Copycat's success, they frequently compare results from multiple Copycat runs with answers given by people to the same questions.

I will sketch a genetic algorithm that could search through the parameter space to find a vector of parameters for its pre-coded concepts and codelets that would lead it to find solutions more often that have a lower temperature and that match human choices.

*Genotype*

The genotype of the GA will be a vector of the parameters being tweaked. Parameters relating to families of codelets and related areas of the architecture would be located next to each other. Approximate upper and lower bounds could be set for some parameters, to keep them in line with human intuitions, and to try and preclude the system from choosing some peculiar combination that appears to work for the particular training set, but performs poorly on novel data.

The full list of parameters would be huge, since Copycat employs a huge array of fudges in all of its computations, especially the codelets, but a partial list should at least include the following:

- the association strengths between concepts in the Slipnet
- conceptual depths
- codelet urgencies
- workspace structure happiness and salience values
- the algorithm for calculating the temperature based on all of the above

*Phenotype*

The phenotype is a version of Copycat running with the parameters in its genotype. Ideally, because Copycat's processing is highly stochastic, it should be run many times on each problem.

*Population*

The easiest population to start with would centre around the set of parameters that the published version of Copycat employs, since these parameters are pretty close to the optimal location in parameter-space for modelling human performance. Having larger variation within the starting population or starting with randomly-generated populations might indicate whether the parameters converge towards a limited number of optima, and to see whether the hand-coded ones can be improved upon.

*Inheritance operators*

A GA might work reasonably well for this problem since it could allow for the majority of the traversal through the space to be performed by crossover recombination, while individual parameters could be tweaked slightly by mutation. Once a set of parameters for a concept/codelet have stabilised, crossover will combine them with other subsets of the parameter-vector that work too.

*Fitness function*

There are various fitness functions that could be employed:

1. The simplest would be to set the fitness as the proportion of people who gave a particular answer to a given letterstrings problem. For instance, if 90% of respondents think that for the problem:

   *abc : abd :: ijkk : ?*

   that *ijll* is the best solution, then that could be given a fitness of 0.9. This could be continued for each answer that people gave, where some very rare answers will have tiny fitness. Answers that no human gave will have a fitness of zero. There would be no negative fitness.
2. The above fitness function would work moderately well, but it would have the unfortunate effect of biasing the system to find common solutions, which may not necessarily be the most *satisfying*. As mentioned earlier, often people will be shown a solution that hadn't occurred to them, which they will then acknowledge to be more satisfying (though less obvious) then their own. To compensate for this, we could:

a) Present subjects with a pre-prepared shortlist of solutions in multiple-choice format, including the less common but more satisfying ones. Subjects would be asked to choose the most satisfying. There would need to be procedures for adding new unconsidered answers to the list occasionally, if a subject discovers an unlisted solution.

b) If everyone chooses the same favourite solutions, then this may lead to too sparse a data set for the second-best and mediocre solutions. The search through the high-dimensional parameter space would require fewer data points (i.e. fewer subjects and questions) if there are various graded fitness-values, rather than one correct answer for each problem and practically no fitness-values for any of the slightly less satisfying answers. This could be solved by having subjects rank all of the solutions. Some scoring system would have to be devised, such that higher-ranked solutions are worth rather more than lower-ranked solutions when totalling up the frequency with which each solution is chosen (like in Formula One Grand Prix championship points).

3. Alternatively, subjects could be asked to assign subjective satisfaction scores (out of 100) to solutions, either their own or chosen from a shortlist. The fitness here could be calculated as:

$$F = 1 - \frac{|S - (T - 100)|}{100}$$

where $F$ is the *fitness*, $S$ is the *average human subjective satisfaction score*, and $T$ is the Copycat *temperature*[2] for that run. When $S = T$, $F = 1$. For a maximum discrepancy, where $S = 100$ and $T = 0$ (or vice versa), $F = 0$. We can imagine various other fitness functions where the fitness might be non-linearly related to the difference between $S$ and $T$, but this illustrates the idea.

This approach might be interesting but problematic. This scheme assumes that there is a consensus about which analogies people find satisfying, since Hofstadter implies this in his discussion. However, without having access to experimental data, it's difficult to know how large the variance between people's subjective assessments would be, but with a little instruction it seems reasonable to hope that this could work.

These fitness functions are just intended to give a flavour of how such a system for tweaking the parameters to give human-like performance might work – there will almost certainly be even better ways of calculating the fitness, discoverable through experiment and differing from domain to domain

Given that Mitchell has written a book on GAs, I presume that this approach has occurred to their group, although I'm not aware of it ever having been implemented.

It is worth noting that if there was some fixed and versatile means of calculating the temperature, then human subjects would not be needed at all. Instead, the system could use its self-calculated temperature as the fitness score, and it could run itself many times in an effort to find a set of parameters that commonly produces low temperatures across its training set. Unfortunately, since the temperature is in part calculated by the activity of the codelets, it cannot be both a dimension in the search space as well as the fitness by means of which the search is directed.

[2] Copycat's temperature parameter ranges from 0-100. We need to calculate Copycat's satisfaction as *100 – temperature*, since a lower Copycat temperature signifies a higher satisfaction.

## Remembering and self-watching

The next important component that needs to be considered is Metacat's capacity for remembering, self-watching and self-evaluation. Although I criticise these mechanisms as being somewhat limited, it is worth noting in Metacat's defence that "the focus in Metacat [was] not on learning to make 'better' analogies, or to make them more 'efficiently', but rather on being able to explain why one analogy is judged to be more compelling than another" [Marshall, 1999].

As described, Metacat produces a trace of every run, which is really a high-level abstraction of the events that occurred during that run. This makes it possible, in principle, to search through past runs to see how they might be similar to the current run at a high-level, even though the actual letterstrings involved may be superficially very different. This is what Carbonell et al. term a 'derivational analogy' [Carbonell, 1986]. Unfortunately though, Metacat doesn't seem to quite do this. As far as I can tell, the search through stored runs is limited to those which involve some of the same letterstrings. This amounts to little more than an engineering hack to avoid falling into already-experienced traps and to allow the reuse of past solutions as a time-saver, although it does allow a basic comparison of problems to identify which particular steps or themes (i.e. instantiated concepts) were present in one and missing from the other.

This approach is restricted in terms of what it can do:

1. It can't form meta-analogies
2. It can't use the parallel terraced scan to compare problems

Over the course of the rest of the paper, I will propose extensions to the architecture that would hopefully address both these issues, and drastically augment the representative power of CuriousCat as a result.

### Meta-analogies

Forming a meta-analogy is not quite as silly as it sounds, and I will describe an example to demonstrate that people can do this, perhaps even with relative ease, and that it might prove a powerful cognitive mechanism. I discussed the example of:

*abc : abd :: xyz : ?*

above. I described how the most satisfying solution is usually considered to be *wyz*. Interestingly, we can see that if we tweak the original letterstrings slightly, the subtle pressures that lead to *wyz* are no longer exerted. Consider:

*rst* : *rsu* :: *xyz* : ?

The same impasse of trying to find the successor to 'z' still applies here, but the appeal of recasting *xyz* as a leftwards predecessorship group is considerably reduced, because the symmetry with *abc* as a rightwards successorship group is enhanced by the fact that the leftmost letter 'a' is the first letter of the alphabet, and the *right*most letter 'z' is the *last* letter of the alphabet. Because this aspect of the symmetry is missing between 'r' and 'z', *wyz* is no longer considered to be so subtle, satisfying and appropriate – indeed, no such single, highly-satisfying solution exists for the *rst* version of the problem.

If we were to take another pair of similar analogy problems, where one has a deeply satisfying solution and the other doesn't, despite the only change being the starting letter of the groups (or some other seemingly trivial and superficial modification), then I think it would be fair to see that this analogy between analogies is a meta-analogy. Moreover, this is not particularly difficult for humans to represent, but Metacat cannot manage it. I will give a very quick example [for a considerably more detailed exposition of the following problems, see Hofstadter, 1995; or Mitchell, 1993]:

*abc* : *abd* :: *mrrjjj* : ?

*ijk* : *ijl* :: *mrrjjj* : ?

In the case of the *abc/mrrjjj* problem, the letter-category successorship group of *abc* is mapped onto a string-length successorship group of *mrrjjj* – that is, a group with first one letter then two letters then three letters. Replacing the rightmost group with its successor in this case is to replace it with a group that is longer by one, i.e. 'jjjj'. Thus, the most satisfying answer to the *abc/mrrjjj* problem is considered to be *mrrjjjj* (*see Fig. 3*). Further, note that:

| | | |
|---|---|---|
| *a* (*1*st letter in the alphabet) | → | *m* (length *1*) |
| *b* (*2*nd letter in the alphabet) | → | *rr* (length *2*) |
| *c* (*3*rd letter in the alphabet) | → | *jjj* (length *3*) |

In the case of the *ijk/mrrjjj* problem, *mrrjjjj* is a less satisfying answer, because although *ijk* contains a successor group, 'i' is not the first, 'j' is not the second and 'k' is not the third letter of the alphabet. Again, this tweak of changing the starting letter from 'a' to 'i' results in a considerably less satisfying solution overall, although the themes of 'successorship', 'sameness' and 'length' are common to both problems. The meta-analogy that I am proposing is then of this form:

*abc/xyz* : *abc/mrrjjj* :: *rst/xyz* : *ijk/mrrjjj*

As an aside, I still get a nosebleed every time I try and decide whether people can manage meta-meta-analogies, and whether this might be useful. Suggestions or tissues would be welcome.

## Using the parallel terraced scan to compare problems

Metacat's means of retrieving past cases appears to be very limited. In the terminology of case-based reasoning [Leake (1996), Kolodner], its retrieval is indexed by the letterstrings and by themes, and not at all by the structure of the trace. What if we wanted to find a case that had similar themes to the one being considered, but used entirely different letterstrings, and also involved lots of snags and dismantling but eventually found a satisfying solution? Metacat couldn't conduct this search, though we might well want it to. After all, what could be more useful than to be reminded of a superficially different but thematically similar problem which was also problematic, but eventually proved tractable? The way to do this would be to conduct a parallel terraced scan on stored memories to tentatively suggest a number of potential candidates, and then winnow down to the particular cases whose theme *and* trace structures match most deeply. This is similar to what Gentner and Markman [1997] refer to as the 'many are called but few are chosen' principle.

## Working on traces in the same way we work on letterstrings

The lesson from the discussion of meta-analogies is that really powerful and abstract thinking requires the ability to further chunk relations and transformations of already-chunked representations. Metacat cannot do this. Its Slipnet is fixed in size and repertoire, and the Slipnet nodes are internally structureless. It cannot chunk events/themes in the trace to compare traces at a higher level of description.

The first major step towards addressing this would be to treat the events (such as 'snag', 'drop in temperature') that are stored in the Metacat trace just like 'event-letters' in a meta-Workspace that we might term the 'Trace-Workspace', along with an accompanying Trace-Slipnet and Trace-Coderack too. Events could then be chunked together to form event-structures of different types, such as:

the 'loop', when the same snag is experienced repeatedly, and no huge drop of temperature results (which would indicate that a solution had been found)

the 'destructive rage', involving a flurry of activity from the dismantler codelets 'frustration', where the same snag is experienced repeatedly, followed by a destructive rage

the 'paradigm shift', composed out of some snags, subsequent dismantlings, some reassembly and a huge drop in temperature

Of course, many many more will exist, if we want to catalogue the various types and combinations of events. Interestingly, if we were to try and represent these event-structures as letterstrings, they might look something like this (using capitals to distinguish them from the standard letterstrings):

| | |
|---|---|
| loop (*L*): | *SSS…* |
| destructive rage (*R*): | *DDD…* |
| frustration (*F*): | e.g. *SSSHDDD*<br>or just: *LHR* |
| paradigm shift (*P*) | e.g. *SSSHDDDBC*<br>or just: *FBC* |

where:

*S* – snag

*D* – dismantlement

*B* – building a structure

*C* – large drop in temperature (colder)

*H* – large increase in temperature (hotter)

This simple (and ugly) notation is intended to illustrate a few important points.

Firstly, notice that the loop and destructive rage look just like special kinds of sameness-groups. *SHD* and *LHR* look almost like successorship groups (though allowing the same letter to have more than one potential successor). *D* and *B*, and *C* and *H*, are opposites. *C* is strongly associated with the rightmost position. If we think of the events as letters, then we find that our letterstring concepts start to apply. This feels like an exciting, though perhaps somewhat obvious-seeming, observation. Given this pleasing applicability of letterstrings concepts to our Trace-architecture, it make much more sense to implement the Trace-Workspace as simply an area within the standard Workspace. Events would be represented at the same level as letterstrings, Trace-concepts would interact with the standard Slipnet-concepts in the same semantic network (although the two groups would probably be fairly sparsely inter-connected), and Copycat codelets could operate upon Trace-structures (though probably not vice versa). This approach could well prove to have powerful advantages, especially with regard to the discussion below about the generation of concepts (fresh or compound). For the moment though, I will continue to talk as though the Trace-architecture is kept separate from the standard architecture for simplicity of exposition.

Secondly, we need a better means of writing down a schema or template for structures. We want to say that a loop has some indefinite number of snags, perhaps with other events sandwiched in between – in other words, we want to say *(S.)**, using the powerful notation of regular expressions. In fact, I will discuss below how the regular expression notation could be utilised as a means of representing structures and codelet algorithms across the Copycat letterstrings domain.

Thirdly, we now have a high-level description that might constrain our trawl through memory when trying to find past problems that usefully resemble the current one. Metacat already looks for problems that have similar themes to the current one, such as 'symmetry', 'successorship', 'predecessorship', 'first letter of alphabet' and 'end of alphabet in the case of the *abc/xyz* problem. It can tell that the *rst/xyz* version lacked the 'first letter of alphabet' and 'symmetry' themes, which is why it's different. Now, perhaps CuriousCat could seek as well for past problems that match these thematic descriptions, as well as having a paradigm shift (for instance) somewhere along the way. It could also see whether all the remembered problems similar to the *rst/xyz* version involved frustration without a paradigm shift, as a means of deciding that further effort on a problem which consistently appeared to have no satisfying solution would be fruitless.

Fourthly, I don't think it's entirely a coincidence that very emotive words like 'frustration' and 'rage' seem to so aptly describe the trace-patterns described here. Being stuck (when there are few particularly urgent codelets waiting and few salient structures calling for attention), feeling encouraged (a series of small drops in temperature) and resignation (when a problem is deemed intractable) are other emotional states that we can easily imagine might fit into this Trace-based language of the emotions. Might even humour be partially describable as a slightly far-fetched or unusual, incongruous or unexpected trace-pattern? As we will see later, I also think that curiosity can be seen partly in these terms.

Finally, the incorporation of the Trace-architecture would go some way towards allowing CuriousCat to see the meta-analogy described above, where we are comparing two pairs of analogies, in which a single superficial-seeming tweak (to the starting letter) has caused one problem to be considerably more frustrating than another.

Seeing such a meta-analogy requires seeing that:

a) they share some similar themes, but the *rst/xyz* version lacks some

b) all of the runs on both problems involve lots of snags, dismantling and rebuilding, but one problem (*abc/xyz*) occasionally admits a highly satisfying solution while the other (*rst/xyz*) never does

c) seeing that both these points also hold true of the second pair of problems, *ijk/mrrjjj* and *ijk/mrrjjjj*

We would be using a parallel terraced scan to do a search through the traces themselves, in order to appreciate that each pair of problems consists of a slight dissimilarity in themes resulting in a drastic difference in how satisfying their best solutions are. Of course, if we were now to keep a trace of the activity on the Trace-Workspace, a 'Trace-Trace', then it would be possible to search through past meta-analogy cases to find similar meta-analogies to the one above.

The architecture I have sketched above does not go into enough detail to show exactly what kind of trace-concepts, codelets and rule-transformations would be needed, nor exactly how the trace and theme information for each problem would be represented on the Trace-Workspace. But it is hoped that it does show how the natural and elegant extension of the Metacat architecture of treating trace data at the same level and with the same mechanisms as the letterstrings could dramatically increase the power and abstraction of its analogy-making.

## Proposing new analogies

Before going any further, I'm going to make what will seem like a digression in order to tie up a thread that we will need for our grand knot later. I'm going to muse about how CuriousCat might perhaps propose its own analogies.

The simplest way to do this would be to start by retrieving a problem from memory, This should ideally be one with a satisfying solution, indicating that it had been fully understood and that all the requisite concepts were available. Then the system could simply fire a rule-transfer codelet at the corresponding structure in all four strings, and see if the analogy still holds. In this way, it could start to see what sort of transformations preserve a particular analogy, and which destroy or undermine it. This subtle comparison of analogies is another way of seeing exactly the kind of thing we were doing with our discussion of meta-analogy earlier.

Alternatively, the system could start with a blank Workspace, and enter a special mode where builder codelets are run on empty space, generating placeholder structures without any letters in them. Then, we could randomly choose a starting letter (biased perhaps towards 'a' and 'z'), and the rest of the string should then be deterministically generable. Finally, a stochastically chosen rule-transfer codelet $R_1$ would generate a transformation from *A* to *B*, another rule-transfer codelet $R_2$ would transform from *A* to *C*, and then $R_1$ would be run again on *C* to generate *D*.

Neither of these methods have been adequately fleshed out, but I feel that this cursory outline serves to show that proposing new analogies should not be an especially difficult problem. Proposing *interesting* analogies based around a theme is, of course, a more fiendish business, but one that we'll set aside.

## Expectation

In his roadmap for Magnificat, Roberts [unpublished] introduces the notion of 'expectation', which I think will prove especially relevant to making CuriousCat curious. It's clear that expectation-violation is one of the triggers for curiosity – when I'm surprised, I get curious about why my predictions were wrong.

Two parts of the dictionary [NSOED] definition of expectation caught my eye: "The state or mental attitude of expecting something to happen" and "Grounds for expecting; especially prospects of inheriting wealth". Expectation is a state, it has intentionality (i.e. you expect *something*) you have grounds for this expectation, and often it's related to the prospect of good things to come. This doesn't help us a great deal.

Instead, I tried to characterise expectation in a much barer form in terms of the Copycat architecture: *an expectation is a top-down influence that directs bottom-up processing to look for something specific that would lead to a drop in temperature if found*. This is by no means a rigid or full definition, and I'm sure it could easily mislead us if taken too seriously.

Roberts' discussion of it is short and tantalising. He proposes a "set of codelets operating simultaneously on the Workspace, comparing and contrasting, building expectations and tearing them down: building brand-new structure" and a recall process, which compares the Workspace with his version of a long-term conceptual memory:

> *"[pulling structure] into the Workspace bit by bit, as urgency demands. If an instance of recall is particularly powerful (it matches structure and fulfils expectations well, thus resolving questions) then its urgency will cause it to proceed rather quickly and completely, but if a memory fits a situation only vaguely, then it will influence the structure in the Workspace only vaguely"*

> *"An expectation can be seen as a scan in progress – it wants to be fulfilled with something, be that something additional*

> *structure in the Workspace, or structure it builds in the Workspace, or structure copied from the [LTM]. I expect the expectation to be a rather powerful organizing force in Magnificat's operation."*

I can't really do much better than that, but I want to try and be more specific. If we reify this expectation on the Workspace as a kind of placeholder structure waiting to be instantiated, we can treat a failure to find part of the letterstrings to fit into it as an violation of this expectation. In a way, the weak bond-structures that get tentatively formed then get quickly dismantled when they interfere with a much stronger structure, or the bridges between strings that don't quite fit all of the structures on each side and get broken down are also expectations that get violated. The difference though is that these are instantiated structures that get built in response to things that are known to exist on the Workspace, whereas the placeholder structures are only there because the recall process has found past or analogous memories that indicate they might be. In either case, Copycat response would:

- raise the temperature (making the whole system's functioning more stochastic)
- direct processing towards those areas of the Workspace to try and resolve the issue

In other words, it will become curious about them. Furthermore, CuriousCat will be able to:

- look for cases in its memory that might be less obviously applicable (i.e. more abstract, perhaps)
- flag the expectation-violation in its trace

and eventually:

- try and form a new concept to plug the gap in its conceptual repertoire.

It is to this last vital and difficult process that I now turn.

## Forming new concepts and codelets

The area that I've spent longest with least reward considering is the issue of how to automate generation of the Slipnet and its associated codelets. This is, in its fullest sense, the AI complete problem of mechanistically and efficiently generating compact hypotheses/categories that capture all and only the features that identify a given set of examples. If we consider analogy-making to be the business of using low level concepts to build high-level concepts that relate situations, then having the right set of low-level concepts for each domain is crucial, as is the ability to generate new concepts should some unexpected aspect of the situations prove to be the essence of the analogy. Indeed, we can see all learning in terms of *forming concepts and applying them to perception in such a way as to generate useful behaviour*.

For a concept to exist in the Slipnet, it requires an associated family of codelets that do the detection, structure-building and -evaluating, dismantling, rule-translating etc. In order to generate a new concept, the node has to be inserted into the Slipnet, assigned a conceptual depth, associated with other concepts, and the whole family of codelets has to be generated. If we see the structures in the Workspace as tokens of concept types in the Slipnet, then generating new structures from the Slipnet nodes and associated codelets is comparatively trivial.

Adding a node to the Slipnet can be done in one of at least a few ways. It can be generated afresh, assigned a default conceptual depth and connected weakly to every single other concept. We could then use some hill-climbing search again to try and search through its parameter space, as before when tweaking all of the system's parameters with the GA. If this method were to prove necessary, it would be worth thinking about how the system could learn after each new problem is presented, perhaps by doing its Slipnet tweaking offline. A second method would be to copy and paste an existing similar concept, mirroring its conceptual depth and associations to other concepts. This is difficult though, because it requires you to figure out which of your current concepts the new, mysterious concept is most similar to. Ideally, we would want to be able to create compound concepts out of combinations of current concepts, or even splice together aspects of two concepts. To do this would require nesting concept-nodes in the Slipnet.

Next, we need to generate new accompanying codelets. We can see two major groups of codelets: those that deal with structures (whether within or between strings), and those that take the rule found that transforms from A to B, and adapt or apply it to C to produce D.

$$A : B :: C : D$$

Generating new rule-transfer codelets seems a particularly intractable problem. Let me try and explain why. Let us imagine that we wanted a 'mirroring' concept, that takes a structure and placed a copy of the reverse-string to its right. Let us imagine that we already have an algorithm for the seeker codelets, so they can tell when they've found one – once we've identified a string as instantiating a concept, we can also build the concept as a structure and we can dismantle it. However, the algorithm that applies the mirror-rule to a part of the string in A to produce B, and in C to produce D is still highly problematic. The only mechanistic method I could devise to be sure to eventually capture the right rule-transfer algorithm would be to do an exhaustive breadth-first search through the source code of some Turing-complete language. For instance, we could take as our alphabet the handful of operations that allow us to define any Turing machine (move tape left, read off digit etc.), and starting with the strings of length 1, build longer and longer strings until we found a Turing machine that performed our rule transfer. We have all the representational power we could possibly want with this method, but it's wholly useless as a realistic implementation solution. The only solution may be to devise a domain-specific high-level language for each domain of rule-primitives like 'move letter in position X to position Y', 'replace letter X with …' etc. Unfortunately, this brings a human back into the loop, and so in a sense is admitting defeat. Perhaps there is a way to deal with the problem of generating rule transfer codelets in terms of the seeking and building codelets, which I think are a little easier to think about.

I will propose a possible letterstrings implementation for the structure-codelets because it's reasonably neat, and might grease our intuition about the kind of complementary language we would need for rule-transfer codelets. The proposal is to use some variant of regular expressions to capture all and only the letter combinations that instantiate a particular concept. To give two examples, sameness would look something like *(a*|b*|c*|d*|…)* and successorship might look like (ab|bc|cd|de|ef|…), where obviously '…' is a technical symbol for my laziness. In the case of successorship, we would have to build successorship groups of length greater than 2 by nesting small ones. This would slightly change things, and would probably require tweaking the successorship parameters to make them easier to build in some way in order to be sure that the system notices all of a long successorship group. This problem highlights the fact that regular expressions are limited in the kinds of string combinations that they can represent, and so this is not a fully satisfying solution. Of course, in the worst case we could restrict the length of our strings to some finite number, and then enumeration would always be an option.

In fact, the possibility of huge enumerated lists that supposedly capture some concept could actually prove a problem. We want to capture the best regular expression for a given concept, i.e. the one that captures all and only the strings that instantiate that concept, using as few enumerations as possible. One way of intuitively understanding this 'minimum description length' (MDL) (Rissanen, 1978) is as the optimal, most compact compromise between a huge list of examples/exceptions, and a single, very lengthy rule that fully captures all the data. Fass and Feldman [2002] discuss how they were able to use the MDL as an indicator of the subjective difficulty of learning some given category. The MDL two-part code will be maximally compact when the following equation is minimised:

$$-\log P(D|H) - \log P(H)$$

where $D$ is the data and $H$ the category hypothesis. The MDL is thus intended to capture "all the data, including the uninformative noisy data that isn't generated by the models" [Rissanen, 1999]. It might be interesting for our purposes to be able to tip the scales of the trade-off between model complexity and data complexity when representing different types of concept.

An alternative approach would be to ignore regular expressions, but still use the idea of evaluating the power of a concept-representation by its minimum description length. We could instead use some sort of substitutional/dictionary encoding [Hankerson et al., 2003], where the system tries to choose as compact as possible a codebook that losslessly encodes all of the strings encountered so far. The serious and unavoidable problem with this method is that it doesn't work well with systematic but sparse data. So, even if the system had learned the successorship concept for the letters it commonly encounters but had never seen a letterstring using the letter 'p' before, none of the learned concepts would have 'p' in their codebooks, and it would not be true at all to say that the system had really understood the concept.

Finally, it's worth noting that Roberts proposes that a suite of generic codelets could be devised that would prove applicable to various domains. This would be great, if possible. His approach is to devolve much of the information from the codelet algorithms to the long-term conceptual memory that he intends to replace the Slipnet and its concept-nodes. These LTM-nodes will have internal structure, and will be composed out of other LTM-nodes. Though interesting, I don't see why it would be any easier to build up the internal structure of an LTM-node than it would be to have codelet algorithms with internal structure, which is one way to see the above proposals. In short, I don't know how his proposal makes the business of generating arbitrary and complex new concepts easier. My only thought is that if we can find a powerful enough representation for the codelet algorithms in the letterstrings domain, we might try and encode other domains in terms of letterstrings. Unfortunately, I don't think this would work so well in domains with continuous rather than discrete perceptual atoms (e.g. some real-number-strings version of *Seekwhence* – see Hofstadter, 1995, ch 1).

The two central problems of forming new concepts are:

- how to tell when a new concept is needed

- how to tell what it should do

I have tried to address the second point, but the first remains. How do you realise when you have a gap in your knowledge that would be useful to fill in? In fact, we answered this question earlier, when we discussed expectation-violation. Forming a new concept was the last in a chain of increasingly drastic options to be undertaken in the face of a series of stubborn expectation violations. We can see almost all of the discussion so far of architecture extensions as being instrumental in helping the system tell when a new concept is needed and what it should do.

Let us imagine that CuriousCat has been presented with a number of problems, for which it was unable to find any solutions whatsoever (or perhaps very unsatisfying ones) for some small proportion. Every time it encounters another unsolvable problem, it flags the expectation-violations and failure in its trace, and tries rebuilding in different ways, and starts various trawls through its memory based on redescriptions of the current situation. It decides to try looking for some concept that will help with a number of these problems, and so uses the unsolvable ones to focus its training set. It tries proposing new analogies based around these problematic cases, and tries to see if it can solve them itself. If it can, then they might provide clues about how to solve the problematic ones. Or, it might indicate that something about the transformations added to generate the new analogies affected the missing concept in some way, and rendered it either unnecessary or tractable. Noticing which themes are added and missing in all of these cases, should provide a good indication of which concepts are similar to the prospective concept.

Of course, the previous discussion assumes that there is only one new concept waiting to be discovered. If there are two or more, the new concept will end up as some sort of amalgam of them all. To resolve this, we need further mechanisms for splitting concepts into two. For this, we could use the same architecture-extensions to pay attention to the frequency with which a pair of concepts are fighting to be instantiated simultaneously as a structure on the same letters. This covariance might indicate an overlap.

Now, I want to return to my early definition of a curious machine as one that *interrogates its environment*. CuriousCat can be seen as engaging in an interative dialogue with the environment about what is required of a new concept. When it starts to feel that a new concept is necessary, CuriousCat can probe the boundaries and situations in which the concept applies by proposing meta-analogies, i.e. by asking 'is this case like this case'? If we allow it a reward signal for the strength of the meta-analogies, the hope is that could learn a great deal in a short time by constraining the space in which the new concept applies.

# Applications

I had very much hoped to discuss how CuriousCat would deal with being ported to two new domains (chess, and something similar to Evans' original geometric puzzles), and how well I thought it would be able to manage without extensive human hand-holding, but unfortunately time and space preclude this.

# Conclusions

## Mapping and transformation

As may have become apparent, I have focused far more on the problem of mapping than transformation. For instance, in the discussion on traces and meta-analogies, I proposed a means of retrieving past situations that might be related in an abstract way, but I hardly mentioned how we might use this knowledge, other than to occasionally concluding when to give up on clearly fruitless problems. Having found a past case that's analogous to this one, we want to see how the solution found there guides the search for a solution to the current problem. For instance, having found the meta-analogy between *abc/xyz//rst/xyz* and *abc/mrrjjj//ijk/mrrjjj*, we want to draw some conclusions about the kinds of tweaks that do and don't affect how satisfying a problem's solution is. In the terminology of case-based reasoning, I have focused upon the case-based remembering, rather than case-based adaptation.

This is partly because one of my primary motivations was to think about how a chess program could aid a human player by presenting analogous examples from past games, so that the human player could see how they unfolded and adapt his game plan accordingly. The responsibility for modifying the retrieved cases lay squarely with the human, since this is a somewhat different and very difficult problem far beyond the scope of this discussion.

To some degree, this is also because the lessons learned from Copycat about fluid concepts scale well, while the Metacat architecture is less rich as a source of ideas about traces, cases and episodic memory. To some degree, I think the architecture extensions proposed here might support this further task of adaptation. However, much more work is needed.

## Relations to a richer definition of curiosity

At the beginning, I gave a rich characterisation of curiosity along the following lines:

> *Being curious is about seeking knowledge that you don't even know for sure that you'll need. It's proactive, requires a learning system and cognitive architecture complex enough to subserve goal-directed and flexible behaviour, recognition of novelty, and some degree of (self-)evaluation. The more complex and diverse the goals, behaviours and representations, the more complex the curiosity manifested. There can be different types of curiosity, triggered under different circumstances, suited to different domains, goals or learning styles.*

I don't believe that CuriousCat would be 'curious' in a truly rich sense, but I felt happy with that characterisation of curiosity when I gave it, and I do feel that the architecture described goes a long way towards it in at least two-thirds of the ways listed. This also serves to make apparent the folly of seeking a single 'curiosity' module, given how many different functions, often originally designed with different goals in mind, were eventually drafted in as integral to the business of being curious. Curiosity just results from the system's methods of learning proactively.

## References

S. Bolland, 'CopyKitten: A Java-Based Implementation and Tutorial of the Copycat Model of analogical thought', Honours thesis, School of Information Technology, UQ, 1997.

Carbonell, J. (1986), 'Derivational analogy: a theory of reconstructive problem-solving and expertise acquisition', in Michalski, Carbonell and Mitchell (ed.), *Machine Learning: an Artificial Intelligence Approach, vol II.*

Chalmers, French and Hofstadter (1991), 'High-Level Perception, Representation and Analogy: A Critique of Artificial Intelligence Methodology' in Hofstadter (1995).

Desai, R., 'Structure-Mapping vs. High-level Perception: Why the Fight is Not Mistaken', in Proceedings of the 19th Annual Conference of the Cognitive Science Society, Stanford, CA, 1997.

Evans, T., 'A program for the solution of geometric analogy intelligence test questions' in Minsky (1968), *Semantic information processing.*

Feldman, J. (2002), 'Simplicity and complexity in human concept learning' 2002 George Miller Award Address, The General Psychologist.

Fass, D. and Feldman, J. (2002), 'Categorization under complexity: a unified MDL account of human learning of regular and irregular categories', in Advances in Neural Information Processing Systems.

French, R. M. (2002). The Computational Modeling of Analogy-Making. Trends in Cognitive Sciences, 6(5), 200-205.

Gentner & Markman (1997), in American Psychologist, vol 52, no. 1, pp 45-56.

Hankerson et al. (2003), *Introduction to information theory and data compression.*

D. Hofstadter, (1995). Fluid Concepts and Creative analogies: computer Models of the Fundamental Mechanisms of Thought. NY: Basic Books.

- (1995). A Review of Mental Leaps: Analogy in Creative Thought." AI Magazine, Fall 1995, 75-80.

Holland, J. H. (1975, reprinted 1992), *Adaptation in natural and artificial systems*, MIT Press.

Kelly, K. (1995), 'By analogy', in Wired, issue 3.11, November.

Kolodner, in Leake (1996), ch 2.

Koza 1997 – for Encyclopedia of Computer Science and Technology, Kent & Williams (ed.).

Langley, Simon, Bradshaw & Zytkow (1987), *Scientific discovery: computational explorations of the creative processes.*

Leake, D. (1996), ed., *Case-based reasoning.*

Marshall, J. B. (1999), *Metacat: a self-watching cognitive architecture for analogy-making and high-level perception*, PhD thesis, Indiana University, Bloomington.

Minsky, *The Emotion Machine* (forthcoming).

- 'Why people think computers can't', online.
- 'Jokes and the cognitive unconscious', online.

Mitchell, M., *Analogy-Making as Perception*, MIT Press, 1993.

Mitchell, M., (1996), *An Introduction to Genetic Algorithms*. Cambridge, MA: The MIT Press.

Morrison, C. & Dietrich, E. (1995), 'Structure-mapping vs. High-level perception: The mistaken fight over the explanation of analogy', in Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society (pp. 678-682).

Rissanen, J. (1978), 'Modeling by shortest data description', in Automatica, 14, 465-471.

Rissannen, J. (1999), Rejoinder, in 'The Computer Journal', vol. 42, no. 4.

Roberts, M. (unpublished), 'A cognitive manifesto', google "Michael Roberts Magnificat".

Sutton, R. and Barto, A. (2002), *Reinforcement learning.*

Veloso, M. 'Flexible strategy learning using analogical replay of problem-solving episodes' in Leake (1996).

Vitanyi, P. and Li, M. (2002), 'Minimum description length induction, Bayesianism and Kolmogorov complexity', online.

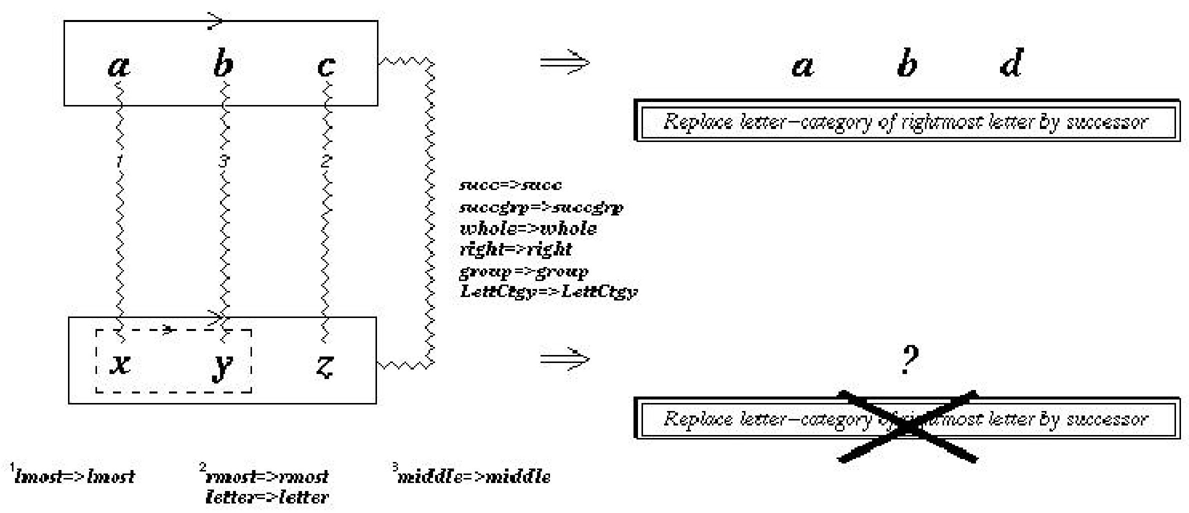
Winston, P. H. (1980), 'Learning and reasoning by analogy', in CACM, vol. 23, no. 12, pg. 689.

# Figures

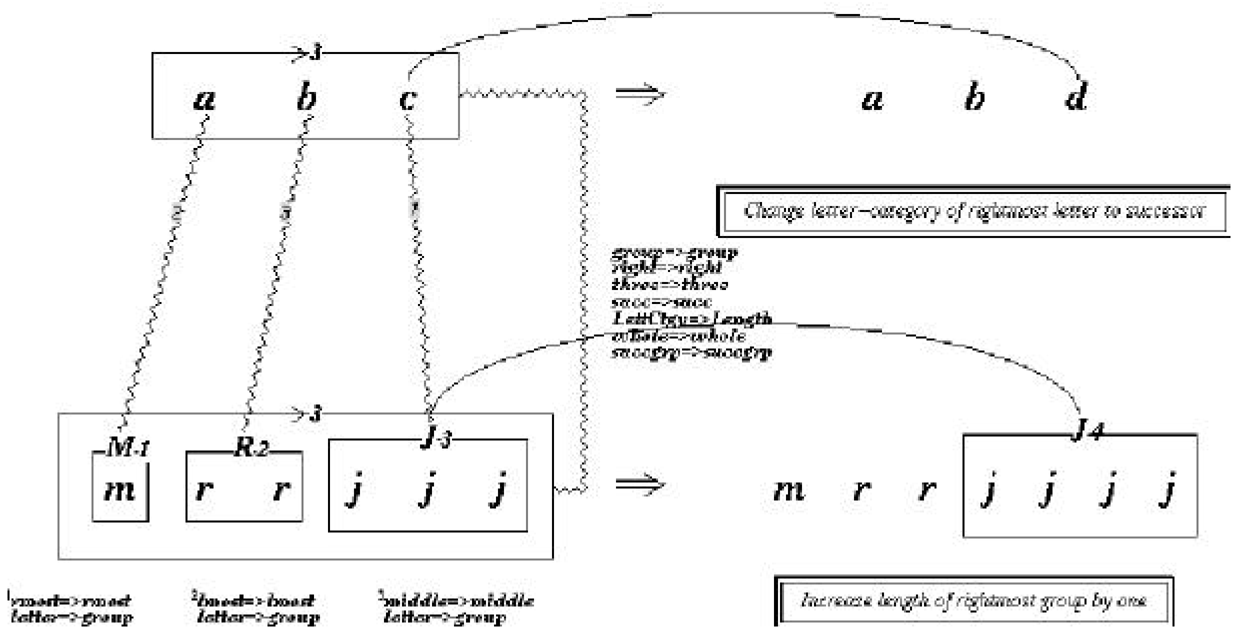
## Figure 1 – Copycat Slipnet

see attached file – ‘copycat slipnet organisation.pdf’

## Figure 2 – Copycat Workspace for the abc/xyz problem



## Figure 3 – Copycat workspace on the abc/mrrjjj problem


a b c
⇒
a b d
Change letter–category of rightmost letter to successor
group=>group
right=>right
three=>three
succ=>succ
LettCtgy=>Length
whole=>whole
succgrp=>succgrp
3
M1
R2
J3
m r r j j j
⇒
m r r j j j j
J4
1rmost=>rmost
letter=>group
2lmost=>lmost
letter=>group
3middle=>middle
letter=>group
Increase length of rightmost group by one