concepts) trying to impose itself on a situation, and a *workspace* pressure as an established viewpoint trying to entrench itself further while keeping rival viewpoints out of the picture. Whereas classical processes are cleanly distinguishable from one another, this is not at all the case for pressures. A given codelet, by running, can advance (or hinder) any number of pressures.

There is thus no way of conceptually breaking up a run into a set of distinct foreordained processes each of which advances piecemeal by being given time slices. The closest one comes to this is when a series of effector codelets' actions *happen* to dovetail so well that the codelets *appear* to have been parts of some predetermined high-level construction process. However, what is deceptive here is that scattered amongst the actions constituting the visible "process", a lot of other codelets — certainly many scouts, and probably other effectors — have played crucial but less visible roles. In any case, there was some degree of luck because randomness played a critical role in bringing about this particular sequence of events. In short, although some large-scale actions tend to look planned in advance, that appearance is illusory; patterns in the processing are all *emergent*.

A useful image here is that of the course of play in a basketball game. Each player runs down the court, zigzagging back and forth, darting in and out of the enemy team as well as their own team, maneuvering for position. Any such move is simultaneously *responding* to a complex constellation of pressures on the floor as well as slightly *altering* the constellation of pressures on the floor. A move is thus fundamentally deeply ambiguous. Although the crowd is mostly concerned with the sequence of players who have the ball, and thus tends to see a localized, serial process unfolding, the players who seldom or never have the ball nonetheless play pivotal roles, in that they mold the globally-felt pressures that control both teams' actions at all moments. A tiny feint of the head or lunge to one side alters the probabilities of all sorts of events happening on the court, both near and far. After a basket has been scored, even though sports announcers and fans always try to account for the structure of the event in clean, spatially local, temporally serial terms (thus trying to impose a *process* on the event), in fact the event was in an essential way distributed all over space and time, amongst all the players. The event consisted of distributed, swiftly shifting pressures pushing *for* certain types of plays and *against* others, and impositions of locality and seriality, though they contain some truth, are merely ways of simplifying what happened for the sake of human consumption. The critical point to hold onto here is the *ambiguity* of any particular action en route to a basket; each action contributes to many potential continuations and cannot be thought of as a piece of some unique "process" coexisting with various other independent "processes" supposedly taking place on the court.

Much the same could be said for Copycat: an outside observer is free, after a run is over, to "parse" the run in terms of specific, discrete processes, and to attempt to impose such a vocabulary on the system's behavior; however, that parsing and labeling is not intrinsic to the system, and such interpretations are in no way unique or absolute, any more than in a basketball game. In other words, a long sequence of codelet actions can add up to what could be perceived, *a posteriori* and by an outsider, as a single coherent drive towards a particular goal, but that is the outsider's subjective interpretation.

The parallel terraced scan

One of the most important consequences of the commingling of multiple pressures is the *parallel terraced scan*. The basic image is that of many "fingers of exploration" simultaneously feeling out various potential pathways at different speeds, thanks to the coexistence of pressures of different strengths. These "fingers of exploration" are tentative probes made by scout codelets, rather than actual events realized by effector codelets. In the Workspace, there is only one *actual* viewpoint at any given time. However, in the background, a host of nearby variants of the actual viewpoint — *virtual* viewpoints — are constantly flickering probabilistically. If any virtual viewpoint is found sufficiently promising by scouts, then they create effector codelets that, when run, will attempt to realize that alternative viewpoint in the Workspace. This entails a "fight" between the incumbent structure and the upstart; the outcome is decided probabilistically, with the weights being determined by the strength of the current structure as opposed to the promise of the rival.

This is how the system's actual viewpoint develops with time. There is always a probabilistic "halo" of many *potential* directions being explored; the most attractive of these tend to be the *actual* directions chosen. Incidentally, this aspect of Copycat reflects the psychologically important fact that conscious experience is essentially unitary, although it is of course an outcome of many parallel unconscious processes.

A metaphor for the parallel terraced scan is provided by the image of a vast column of ants marching through a forest, with hordes of small scouts at the head of the column making small random forays in all directions (although exploring some directions more eagerly and deeply than others) and then returning to report; the collective effect of these many "feelers" will then determine the direction to be followed by the column as a whole. This is going on at all moments, of course, so that the column is constantly adjusting its pathway in slight ways.

The term "parallel terraced scan" comes from the fact that scouting expeditions are structured in a *terraced* way; that is, they are carried out in stages, each stage contingent upon the success of the preceding one, and probing a

little more deeply than the preceding one. The first stage is computationally cheap, so the system can afford to have many first-stage scouts probing in all sorts of directions, including quite unlikely directions. Succeeding stages are less and less cheap; consequently the system can afford fewer and fewer of them, which means it has to be increasingly selective about the directions it devotes resources to looking in. Only after a pathway has been deeply explored and found to be very promising are effector codelets created, which then will try to actually swerve the whole system down that pathway.

The constellation of top-down pressures at any given time controls the biases in the system's exploratory behavior, and also plays a major role in determining the actual direction the system will move in; ultimately, however, top-down pressures, no matter how strong, must bow to the reality of the situation itself, in the sense that prejudices alone cannot force inappropriate concepts to fit to reality. Top-down pressures must adapt when the pathways they have urged turn out to fail. The model is made explicitly to allow this kind of intermingling of top-down and bottom-up processing.

Time-evolving biases

At the very start of a run, the Coderack contains exclusively bottom-up similarity-scanners, which represent no situation-specific pressures. In fact, it is their job to make small discoveries that will then start generating such pressures. As these early codelets run, the Workspace starts to fill up with bonds and small groups and, in response to these discoveries, certain nodes in the Slipnet are activated. In this way, situation-specific pressures are generated and cause top-down codelets to be spawned by concepts in the Slipnet. Thus top-down codelets gradually come to dominate the Coderack.

At the outset of a run, the Slipnet is "neutral" (*i.e.*, in a standard configuration with a fixed set of concepts of low depth activated), meaning that there are no situation-specific pressures. At this early stage, all observations made in the Workspace are very local and superficial. Over the course of a run, the Slipnet moves away from its initial neutrality and becomes more and more biased toward certain organizing concepts — *themes* (highly activated deep concepts, or constellations of several such concepts). Themes then guide processing in many pervasive ways, such as determining the saliences of objects, the strengths of bonds, the likelihood of various types of groups to be made, and in general, the urgencies of all types of codelets.

It should not be imagined, incidentally, that a "neutral" Slipnet embodies no biases whatsoever; it certainly does (think of the permanent inequality of various nodes' conceptual depths, for instance). The fact that at the outset, a sameness group is likely to be spotted and reified faster than a successor group of the same length, for instance, represents an initial bias favoring sameness

over successorship. The important thing is that at the outset of a run, the system is more open than at any other time to *any* possible organizing theme (or set of themes); as processing takes place and perceptual discoveries of all sorts are made, the system loses this naïve, open-minded quality, as indeed it ought to, and usually ends up being "closed-minded" — that is, strongly biased towards the pursuit of some initially unsuspected avenue.

In the early stages of a run, almost all discoveries are on a very small, local scale: a primitive object acquires a description, a bond is built, and so on. Gradually, the scale of actions increases: small groups begin to appear, acquire their own descriptions, and so on. In the later stages of a run, actions take place on an even larger scale, often involving complex, hierarchically structured objects. Thus, over time there is a clear progression, in processing, from locality to globality.

Temperature as a regulator of open-mindedness

At the start of a run, the system is open-minded, and for good reason: it knows nothing about the situation it is facing. It doesn't matter all that much *which* codelets run, since one wants many different directions to be explored; hence decision-making can be fairly capricious. However, as swarms of scout codelets and local effector codelets carry out their jobs, that status gradually changes; in particular, as the system acquires more and more information, it starts creating a coherent viewpoint and focusing in on organizing themes. The more informed the system is, the more important it is that top-level decisions not be capriciously made. For this reason, there is a variable that monitors the stage of processing, and helps to convert the system from its initial largely bottom-up, open-minded mode to a largely top-down, closed-minded one. This variable is given the name *temperature*.

What controls the temperature is the *degree of perceived order* in the Workspace. If, as at the beginning of every run, no structures have been built, then the system sees essentially no order, which translates into a need for broad, open-minded exploration; if, on the other hand, there is a highly coherent viewpoint in the Workspace, then the last thing one wants is a lot of voices clamoring for irrelevant actions in the Workspace. Thus, temperature is essentially an inverse measure of the *quality of structure* in the Workspace: the more structures there are, and the more coherent they are with one another (as measured by their strengths), the lower the temperature. Note that although the overall trend is for temperature to wind up low at the end of a run, a *monotonic* drop in temperature is not typical; often, the system's temperature goes up and down many times during a run, reflecting the system's uncertain advances and retreats as it builds and destroys structures in its attempts to home in on the best way to look at a situation.

What the temperature itself controls is the *degree of randomness used in decision-making*. Decisions of every sort are affected by the temperature — which codelet to run next, which object to focus attention on, which of two rival structures should win a fight, and so on. Consider a codelet, for instance, trying to decide where to devote its attention. Suppose that Workspace object A is exactly twice as salient as object B. The codelet will thus tend to be more attracted to A than to B. However, the precise discrepancy in attractive power between A and B will depend on the temperature. At some mid-range temperature, the codelet will indeed be twice as likely to go for A as for B. However, at very *high* temperatures, A will be hardly any more attractive than B to the codelet. By contrast, at very *low* temperatures, the probability of choosing A over B will be much greater than two to one. For another example, consider a codelet trying to build a structure that is incompatible with a currently existing strong structure. Under low-temperature conditions, the strong structure will tend to be very stable (*i.e.*, hard to dislodge), but if the temperature should happen to rise, it will become increasingly susceptible to being swept away. In "desperate times", even the most huge and powerful structures and worldviews can topple.

The upshot of all this is that at the start of a run, the system explores possibilities in a wild, scattershot way; however, as it builds up order in the Workspace and simultaneously homes in on organizing themes in the Slipnet, it becomes an increasingly conservative decision-maker, ever more deterministic and serial in its style. Of course, there is no magic crossover point at which nondeterministic parallel processing turns into deterministic serial processing; there is simply a gradual tendency in that direction, controlled by the system's temperature.

Note that the notion of temperature in Copycat differs from that in simulated annealing, an optimization technique sometimes used in connectionist networks (Kirkpatrick, Gelatt, & Vecchi, 1983; Hinton & Sejnowski, 1983; Smolensky, 1983). In simulated annealing, temperature is used exclusively as a top-down randomness-controlling factor, its value falling monotonically according to a predetermined, rigid "annealing schedule". By contrast, in Copycat, the value of the temperature reflects the current quality of the system's understanding, so that temperature acts as a *feedback mechanism* that determines the degree of randomness used by the system. Thus, the system itself controls the degree to which it is willing to take risks.

Long after the concept of temperature had been conceived and implemented in the program, it occurred to us that temperature could serve an extra, unanticipated role: the *final* temperature in any run could give a rough indication of how good the program considered its answer to be (the lower the temperature, of course, the more desirable the answer). The idea is simply that the quality of an answer is closely correlated with the amount of strong, coherent

structure underpinning that answer, and temperature is precisely an attempt to measure that quantity. From the moment we realized this, we kept track of the final temperatures of all runs, and those data provided some of the most important insights into the program's "personality", as will be apparent when we discuss in detail the results of runs.

Overall trends during a run

In most runs, despite local fluctuations here and there, there is a set of overall tendencies characterizing how the system evolves in the course of time. These tendencies, although they are all tightly linked together, can be roughly associated with different parts of the architecture, as follows.

- In the Slipnet, there is a general tendency for the initially activated concepts to be *conceptually shallow*, and for concepts that get activated later to be increasingly *deep*. There is also a tendency to move from *no themes* to *themes* (*i.e.*, clusters of highly activated, closely related, high-conceptual-depth concepts).
- In the Workspace, there is a general tendency to move from a state of *no structure* to a state with *much structure*, and from a state having *many local, unrelated objects* to a state characterized by *few global, coherent structures*.
- As far as the processing is concerned, it generally exhibits, over time, a gradual transition from *parallel* style toward *serial* style, from *bottom-up* mode to *top-down* mode, and from an initially *nondeterministic* style toward a *deterministic* style.

The Intimate Relation between Randomness and Fluidity

It may seem deeply counterintuitive that randomness should play a central role in a computational model of intelligence. However, careful analysis shows that it is inevitable if one believes in any sort of parallel, emergent approach to mind.

Biased randomness gives each pressure its fair share

A good starting point for such analysis is to consider the random choice of codelets (biased according to their urgencies) from the Coderack. The key notion, stressed in earlier sections, is that the urgency attached to any codelet represents the estimated proper *speed* at which to advance the pressures for which it is a proxy. Thus it would make no sense at all to treat higher urgencies as higher *priorities* — that is, always to pick the highest-urgency codelets first. If one were to do that, then lower-urgency codelets would never get run at all, so

the effective speeds of the pressures they represent would all be zero, which would totally defeat the notion of commingling pressures, the parallel terraced scan, and temperature.

A more detailed analysis is the following. Suppose we define a "grass-roots" pressure as a pressure represented by a large swarm of low-urgency codelets, and an "elite" pressure as one represented by a small coterie of high-urgency codelets. Then a policy to select high-urgency codelets most of the time would arbitrarily favor elite pressures. In fact, it would allow situations wherein any number of grass-roots pressures could be entirely squelched by just *one* elite pressure — even if the elite pressure constituted but a small fraction of the *total* urgency (the sum of the urgencies of all the codelets in the Coderack at the time), as it most likely would. Such a policy would result in a very distorted image of the overall makeup of the Coderack (*i.e.*, the distribution of urgencies among various pressures). In summary, it is imperative that during a run, low-urgency codelets get mixed in with higher-urgency codelets, and in the right proportion — namely, in the proportions dictated by urgencies, no more and no less. As was said earlier, only by using probabilities to choose codelets can the system achieve (via statistics) a *fair* allocation of resources to each pressure, even when the strengths of various pressures change as processing proceeds.

Randomness and asynchronous parallelism

One might well imagine that the need for such randomness (or biased nondeterminism) is simply an artifact of this architecture's having been designed to run on a sequential machine; were it redesigned to run on parallel hardware, then all randomness could be done away with. This turns out to be not at all the case, however. To see why, we have to think carefully about what it would mean for this architecture to run on parallel hardware. Suppose that there were some large number of parallel processors to which tasks could be assigned, and that each processor's speed could be continuously varied. It is certainly not the case that one could assign *processes* to *processors* in a one-to-one manner, since, as has been stressed, there is no clear notion of "process" in this architecture. Nor could one assign one *pressure* to each processor, since codelets are not univalent as to the pressures that they represent. The only possibility would be to assign a processor to every single codelet, letting it run at a speed defined by that codelet's urgency. (Note that this requires a very large number of co-processors — hundreds, if not thousands. Moreover, since the codelet population varies greatly over time, the number of processors in use at different times will vary enormously. However, on a conceptual level, neither of those poses a problem in principle.)

Now notice a crucial consequence of this style: since all the processors are running at speeds that are completely independent of one another, they are

effectively carrying out *asynchronous* computing, which means that relative to one another, the instants at which they carry out actions in the (shared) Workspace are totally decoupled — in short, entirely random relative to one another. This is a general fact: asynchronous parallelism is inseparable from processors' actions being random relative to one another (as pointed out in Hewitt, 1985). Thus parallelism provides no escape from the inherent randomness of this architecture. When it runs on serial hardware, some *explicit* randomizing device is utilized; when it runs on parallel hardware, the randomness is *implicit*, but no less random for that.

The earlier image of the swiftly-changing panorama of a basketball game may help to make this necessary connection between asynchronous parallelism and randomness more intuitive. Each player might well feel that the snap decisions being made constantly inside their own head are anything but random — that, in fact, their decisions are rational responses to the situation. However, from the point of view of *other* players, what any one player does is not predictable — a player's mind is far too complex to be modeled, especially in real time. Thus, because all the players on the court are complex, independent, asynchronously-acting systems, each player's actions *necessarily* have a random (*i.e.*, unpredictable) quality from the point of view of all the other players. And obviously, the more unpredictable a team seems to its opponents, the better.

A seeming paradox: Randomness in the service of intelligence

Even after absorbing all these arguments, one may still feel uneasy with the proposition that greater intelligence can result from making *random* decisions than from making *systematic* ones. Indeed, when the architecture is described this way, it sounds nonsensical. Isn't it always wiser to choose the *better* action than to choose at *random*? However, as in so many discussions about mind and its mechanisms, this appearance of nonsensicality is an illusion caused by a confusion of levels.

Certainly it would seem extremely counterintuitive — in fact, downright nonsensical — if someone suggested that a melody-composition program (say) should choose its next note by throwing dice, even weighted dice. How could any global coherence come from such a process? This objection is of course totally valid — good melodies cannot be produced in that way (except in the absurd sense of millions of monkeys plunking away on piano keyboards for trillions of years and coming up with "Blue Moon" once in a blue moon). But our architecture in no way advocates such a coarse type of decision-making procedure!

The choice of next note in a melody is a *top-level* macro-decision, as opposed to a low-level act of "micro-exploration". The purpose of micro-exploration is to efficiently explore the vast, foggy world of possibilities lying ahead without getting

bogged down in a combinatorial explosion; for this purpose, randomness, being equivalent to non-biasedness, is the *most efficient* method. Once the terrain has been scouted out, much information has been gained, and in most cases some macroscopic pathways have been found to be more promising than others. Moreover — and this is critical — the more information that has been uncovered, the more the temperature will have dropped — and the lower the temperature is, the less randomness is used. In other words, the more confidently the system believes, thanks to lots of efficient and fair micro-scouting in the fog, that it has identified a particular promising pathway ahead, the more certain it is to make the macro-decision of picking that pathway. Only when there is tight competition is there much chance that the favorite will not win, and in such a case, it hardly matters since even after careful exploration, the system is not persuaded that there is a clear best route to follow.

In short, in the Copycat architecture, hordes of random forays are employed on a microscopic level when there is a lot of fog ahead, and their purpose is precisely to get an evenly-distributed sense of what lies out there in the fog rather than simply plunging ahead blindly, at random. The foggier things are, the more unbiased should be the scouting mission, hence the more randomness is called for. To the extent that the scouting mission succeeds, the temperature will fall, which in turn means that the well-informed macroscopic decision about to be taken will be made *non*-randomly. Thus, randomness is used *in the service of*, and not in opposition to, intelligent nonrandom choice.

A subtle aspect of this architecture is that there are all shades between complete randomness (much fog, high temperature) and complete determinism (no fog, low temperature). This reflects the fact that one cannot draw a clean, sharp line between micro-exploratory scouting forays and confident, macroscopic decisions. For instance, a smallish, very local building or destruction operation carried out in the Workspace by an effector codelet working in a mid-range temperature can be thought of as lying somewhere in between a micro-exploratory foray and a well-informed macroscopic decision.

As a final point, it is interesting to note that non-metaphorical fluidity — that is, the physical fluidity of liquids like water — is inextricably tied to random microscopic actions. A liquid could not flow in the soft, gentle, *fluid* way that it does, were it not composed of tiny components whose micro-actions are completely random relative to one another. This does not, of course, imply that the top-level action of the fluid *as a whole* takes on any appearance of randomness; quite the contrary! The flow of a liquid is one of the most nonrandom phenomena of nature that we are familiar with. This does not mean that it is by any means *simple*; it is simply familiar and natural-seeming. Fluidity is an emergent quality, and to simulate it accurately requires an underlying randomness.

Copycat's Performance: A Forest-level Overview

The statistically emergent robustness of Copycat

Now that the architecture of the Copycat program has been laid out, we can take a tour through the program's performance on a number of problems in its letter-string microworld. As was discussed earlier, Copycat's microworld was designed to isolate, and thus to bring out very clearly, some of the essential issues in high-level perception and analogy-making in general. The program's behavior on the problems presented here demonstrates how it deals with these issues, how it responds to variations in pressures, and how it is able, starting from exactly the same state on each new problem, to fluidly adapt to a range of different situations.[2] (The program's performance on a much larger set of problems and some comparisons with people's performance on the same problems are given in Mitchell, 1993.)

On any given run on a particular problem, the program settles on a specific answer; however, since the program is permeated with nondeterminism, different answers (to the same problem) are possible on different runs. The nondeterministic decisions the program makes (*e.g.*, which codelet to run next, which objects a codelet should act on, etc.) are all at a microscopic level, compared with the macroscopic-level decision of what answer to produce on a given run. Every run is different at the microscopic level, but statistics lead to far more deterministic behavior at the macroscopic level. For example, there are a huge number of possible routes (at the microscopic level of individual codelets and their actions) the program can take to arrive at the solution ***ijl*** to Problem 1, and a large number of micro-biases tend to push the program down one of those routes rather than down one of the huge number of possible routes to ***ijd***. Thus in this problem, at a macroscopic level, the program is very close to being deterministic: it gets the answer ***ijl*** almost all the time.

The phenomenon of macroscopic determinism emerging from microscopic nondeterminism is often demonstrated in science museums by means of a contraption in which several thousand small balls are allowed to tumble down, one by one, through a regular grid of horizontal pins that run between two parallel vertical plexiglass sheets. Each ball, as it falls, bounces helter-skelter off various pins, eventually winding up in one of some 20 or 30 adjacent equal-sized bins forming a horizontal row at the bottom. As the number of balls that have

2. The current version of the Copycat program can deal only with problems whose initial change involves a replacement of at most one letter (*e.g.*, ***abc*** ⇒ ***abd***, or ***aabc*** ⇒ ***aabd***; of course the *answer* can involve a change of more than one letter, as in ***aabc*** ⇒ ***aabd***; ***ijkk*** ⇒ ***ijll***). This is a limitation of the program as it now stands; in principle, the letter-string domain is much larger. But even given this limitation, a very large number of interesting problems can be formulated, requiring considerable mental fluidity. (For a good number of examples of such problems, see Hofstadter, 1984b or Mitchell, 1993.)

fallen increases, the stacks of balls in the bins grow. However, not all bins are equally likely destinations, so different stacks grow at different rates. In fact, the heights of the stacks in the bins at the bottom gradually come to form an excellent approximation to a perfect gaussian curve, with most of the balls falling into the central bins, and very few into the edge bins. This reliable buildup of the mathematically precise gaussian curve out of many unpredictable, random events is fascinating to watch.

In Copycat, the set of bins corresponds to the set of different possible answers to a problem, and the precise pathway an individual ball follows, probabilistically bouncing left and right many times before "choosing" a bin at the bottom, corresponds to the many stochastic micro-decisions made by the program (at the level of individual codelets) during a single run. Given enough runs, a reliably repeatable pattern of answer frequencies will emerge, just as a near-perfect gaussian curve regularly emerges in the bins of a "gaussian pinball machine".

Copycat's "personality" is revealed through bar graphs

We present these patterns in the form of bar graphs, one for each problem, giving the frequency of occurrence (representing surface appeal) and the average end-of-run temperature (representing quality) for each different answer. For each problem, a bar graph is given, summarizing 1,000 runs of Copycat on that problem. The number 1,000 is somewhat arbitrary; after about 100 runs on each problem, the basic statistics do not change much. The only difference is that as more and more runs are done on a given problem, certain bizarre and improbable "fringe" answers, such as *ijj* in Problem 1 (see Figure V–1), begin to appear very occasionally; if 2,000 runs were done on Problem 1, the program would give perhaps one or two other such answers, each once or twice. This allows the bar graphs to make a very important point about Copycat: even though the program has the potential to get strange and crazy-seeming answers, the mechanisms it has allow it to steer clear of them almost all of the time. It is critical that the program (as well as people) be allowed the *potential* to follow risky (and perhaps crazy) pathways, in order for it to have the flexibility to follow *insightful* pathways, but it also has to avoid following bad pathways, at least most of the time.

In the bar graph of Figure V–1, each bar's height gives the relative frequency of the answer it corresponds to, and printed above each bar is the actual number of times that answer was given. The average final temperature appears below each bar. The frequency of a given answer can be thought of as an indicator of how *obvious* or *immediate* that answer is, given the biases of the program. For example, *ijl*, produced 980 times, is much more immediate to the program than *ijd*, produced 19 times, which is in turn much more obvious than the strange answer *ijj*, produced only once. (To get the latter answer, the program decided to replace the rightmost letter by its predecessor rather than

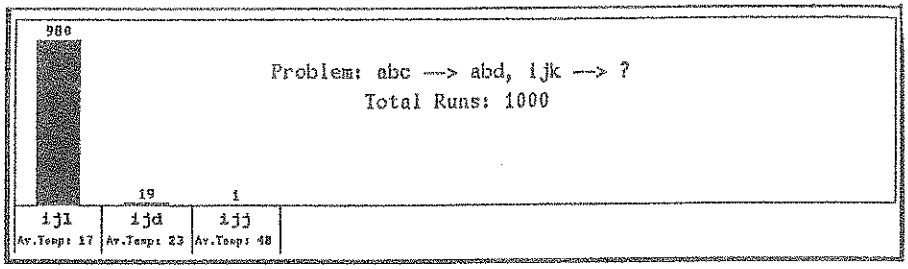

*Figure V–1. Bar graph summarizing 1,000 runs of the Copycat program on the analogy problem "**abc** ⇒ **abd**; **ijk** ⇒ ?".*

its successor. This slippage is always possible in principle, since *successor* and *predecessor* are linked in the Slipnet. However, as can be seen by the rarity of this answer, it is extremely unlikely in this situation: under the pressures evoked by this problem, *successor* and *predecessor* are almost always considered too distant for a slippage to be made between them.)

Although the frequencies shown in Figure V–1 seem quite reasonable, it is not intended that they should precisely reproduce the frequencies one would find if this problem were posed to humans, since, as we said earlier, the program is not meant to model all the domain-specific mechanisms people use in solving these letter-string problems. Rather, what is interesting here is that the program does have the potential to arrive at very strange answers (such as ***ijj***, but also many others), yet manages to steer clear of them almost all the time.

As we said earlier, the average final temperature of an answer can be thought of as the program's own assessment of that answer's *quality*, with lower temperatures meaning higher quality. For instance, the program assesses ***ijl*** (average final temperature 17) to be of somewhat higher quality than ***ijd*** (temperature 23), and of much higher quality than ***ijj*** (temperature 48).

One can get a sense for what a given numerical value of temperature represents by seeing how various sets of perceptual structures built by the program affect the temperature. This will be illustrated in the next section, when a detailed set of screen dumps from a run of Copycat is presented. Roughly speaking, an average final temperature below 30 indicates that the program was able to build a fairly strong, coherent set of structures — that it had, in some sense, a reasonable "understanding" of what was going on in the problem. Higher final temperatures usually indicate that some structures were weak, or perhaps that there was no coherent way of mapping the initial string onto the target string.

The program decides probabilistically when to stop running and produce an answer, and although it is much more likely to stop when the temperature is low, it sometimes stops before it has had an opportunity to build all appropriate structures. For example, there are runs on Problem 1 in which the program

stops before the target string has been grouped as a whole; the answer is still often *ijl*, but the final temperature is higher than it would have been if the program had continued. This kind of run increases the average final temperature for this answer. The lowest possible temperature for answer *ijl* is about 7, which is about as low as the temperature ever gets.[3]

Systematically studying the effects of variant problems

Systematic studies can be done in which a given problem is slightly altered in various ways. Each such variant tampers with the pressures that the original problem evokes, and one can expect effects of this to show up in the bar graph for that problem. For instance, Problem 2, discussed above, is a variant of Problem 1 in which the doubling of letters shifts the "stresses" in the strings *abc* and *ijk*; one might expect this to make the *aa* and the *kk* far more salient and more similar to each other than the *a* and *k* were in Problem 1, thus pushing towards a crosswise mapping in which the two double letters correspond.

In Figure V–2, one sees that despite the pressure towards a crosswise mapping, the "Replace rightmost group by successor" answer (*ijll*) is still the most common answer and the "Replace rightmost letter by successor" answer (*ijkl*) is second, indicating the lingering appeal of the straightforward *leftmost* ⇒ *leftmost*, *rightmost* ⇒ *rightmost* view, even here. However, the pressure is felt to some extent: *jjkk* makes a good showing and *hjkk* has some representatives too, as well as having by far the lowest average temperature. (This is to be contrasted with the results on Problem 1: note that in 1,000 runs, the program *never* gave an answer involving a replacement of the leftmost letter.) The answers on the fringe here include *jkkk* (which is similar to *jjkk*, but results from grouping the *two* leftmost letters — a far-fetched and, to most people, unappealing way of "parsing" the string); *ijkd* and *ijdd* (both based on the rule "Replace the rightmost letter by *d*", but flexed in different ways because of different bridges built from the *c*); *ijkk* (replacing all *c*'s by *d*'s); and *djkk* (replacing the *i* by a *d* instead of by its successor or predecessor).

Another variant on Problem 1 is the following:

3. Suppose the letter-string *abc* were changed to *abd*; how would you change the letter-string *kji* in "the same way"?

Here a literal application of the original rule ("Replace rightmost letter by successor") would yield *kjj*, which ignores an abstract similarity between *abc*

3. There is a problem with the way temperature is calculated in the program as it now stands. As can be seen, the answer *ijd* has an average final temperature almost equal to that of *ijl* (even though it is much less frequent), whereas most people feel it is a far worse answer. This, along with other problems with the current program, is discussed in detail in Mitchell, 1993.

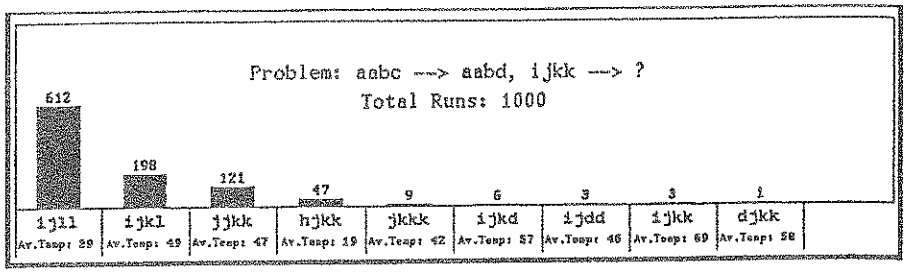

Figure V–2. Bar graph summarizing 1,000 runs of the Copycat program on the analogy problem "aabc ⇒ aabd; ijkk ⇒ ?".

and ***kji***. An alternative many people prefer is ***lji*** ("Replace the leftmost letter by its successor"), which is based on seeing both strings in terms of a *successorship* fabric, in one string running to the right and in the other one to the left; thus there is a slippage from the concept *right* to the concept *left*, which in turn gives rise to the "cousin" slippage *rightmost* ⇒ *leftmost*. Another answer given by many people is ***kjh*** ("Replace the rightmost letter by its predecessor"), in which one string is seen as having a *successorship* fabric and the other as having a *predecessorship* fabric (both viewed as moving in the same spatial direction), thus involving a slippage of the concept *successor* into the concept *predecessor*.

As can be seen in Figure V–3, there are three answers that predominate, with ***kjh*** being the most common (and having the lowest average final temperature), and ***kjj*** and ***lji*** almost tying for second place (the latter being a bit less common, but having a much lower average final temperature). The answer ***kjd*** comes in a very distant fourth, and then there are two "fringe" answers with but one instance apiece: ***dji*** (an implausible blend of insight and rigidity in which the opposite spatial direction of the two successor groups ***abc*** and ***kji*** was seen, but instead of the leftmost letter being replaced by its successor, it was replaced by a ***d***—and yet, notice the relatively low temperature on this answer, indicating that a strong set of structures was built!), and ***kji*** (reflecting the literal-minded rule "Replace ***c*** by ***d***", where there are no ***c***'s in ***kji***), which has a very high temperature of 89, indicating that on this run, almost no structures were built before the program chanced, against very high odds, to stop.

How hidden concepts emerge from dormancy

Consider now the following problem, which involves a very different set of pressures from those in the previous problems:

4. Suppose the letter-string ***abc*** were changed to ***abd***; how would you change the letter-string ***mrrjjj*** in "the same way"?

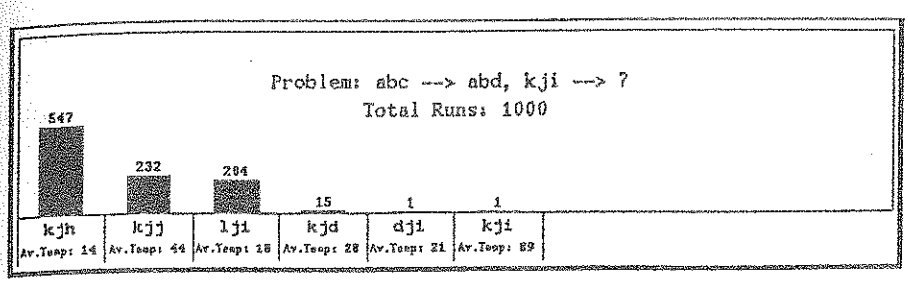

*Figure V–3. Bar graph summarizing 1,000 runs of the Copycat program on the analogy problem "**abc** ⇒ **abd**; **kji** ⇒ ?".*

There is a seemingly reasonable, straightforward solution: ***mrrkkk***. Most people give this answer, reasoning that since ***abc***'s rightmost letter was replaced by its successor, and since ***mrrjjj*** 's rightmost "letter" is actually a *group* of ***j***'s, one should replace *all* the ***j***'s by ***k***'s. Another possibility is to take the phrase "rightmost letter" literally, thus replacing only the rightmost ***j*** by ***k***, giving ***mrrjjk***. However, neither answer is very satisfying, since neither takes into account the salient fact that ***abc*** is an alphabetic sequence (*i.e.*, a successor group). This fabric of ***abc*** is an appealing and seemingly central aspect of the string, so one would like to use it in making the analogy, but there is no obvious way to do so. No such fabric seems to weave ***mrrjjj*** together. So either (like most people) one settles for ***mrrkkk*** (or possibly ***mrrjjk***), or one looks more deeply. But where to look, when there are so many possibilities?

The interest of this problem is that there happens to be an aspect of ***mrrjjj*** lurking beneath the surface that, once recognized, yields what many people feel is a deeply satisfying answer. If one ignores the *letters* in ***mrrjjj*** and looks instead at *group lengths*, the desired successorship fabric is found: the lengths of groups increase as "1–2–3". Once this hidden connection between ***abc*** and ***mrrjjj*** is discovered, the rule describing the change ***abc*** ⇒ ***abd*** can be adapted to apply to ***mrrjjj*** as "Replace the length of the rightmost group by its successor", yielding "1–2–4" at the abstract level, or, more concretely, ***mrrjjjj***.

Thus this problem demonstrates how a previously irrelevant, unnoticed aspect of a situation can emerge as relevant in response to pressures. The crucial point is that the process of perception is not just about deciding which *clearly apparent* aspects of a situation should be ignored and which should be taken into account; it is also about the question of how aspects that were initially considered to be irrelevant — or rather, that were initially so far out of sight that they were not even recognized as being irrelevant! — can *become* apparent and relevant in response to pressures that emerge as the understanding process is taking place.

Sometimes, given certain pressures, a concept that one initially had no idea was germane to the situation will emerge seemingly from nowhere and turn out to be exactly what was needed. In such cases, however, one should not feel upset about not having suspected its relevance at the outset. In general, far-out ideas (or even ideas *slightly* past one's defaults) ought not continually occur to people for no good reason; in fact, a person to whom this happens is classified as crazy or crackpot.

Time and cognitive resources being limited, it is vital to resist nonstandard ways of looking at situations without strong pressure to do so. You don't check the street sign at the corner, every time you go outdoors, to reassure yourself that your street's name hasn't been changed. You don't worry, every time you sit down for a meal, that perhaps someone has filled the salt shaker with sugar. You don't worry, every time you start your car, that someone might have stuck a potato in its tailpipe or attached a bomb to its chassis. However, there are pressures — such as receiving a telephone threat on your life — that would make such a normally unreasonable suspicion start to seem reasonable. (These ideas overlap with Kahneman & Miller's 1986 treatment of counterfactuals, and are also closely related to the frame problem in artificial intelligence, as discussed in McCarthy & Hayes, 1969.)

Not only is pressure needed to evoke a dormant concept in trying to make sense of a situation, but the concepts brought in are often clearly related to the source of the pressure. For example, when one looks carefully at Problem 4 (as we will do in the next main section), one can see how certain aspects of it create pressures that, acting in concert, stand a decent chance of evoking the concept of *group length*. Some critical aspects of the story (not in any particular order) are these: (1) once successor relations have been noticed in ***abc***, there arises a top-down pressure to look for them in ***mrrjjj*** as well; (2) once the ***rr*** and ***jjj*** sameness groups have been perceived in ***mrrjjj***, the normally dormant concept *length* becomes weakly active and lingers in the background; (3) the perception of these sameness groups leads to top-down pressure to perceive other parts of the same string as sameness groups as well, and the only way this can be done is the unlikely possibility of perceiving the ***m*** as a sameness group consisting of *only one letter*; and (4) after standard concepts have failed to yield progress in making sense of the situation at hand, resistance to bringing in nonstandard concepts decreases.

People occasionally give the answer ***mrrkkkk***, replacing *both* the letter category *and* the group length of the rightmost group by their successors (***k*** and 4, respectively). Despite its interest, this answer confounds aspects of the two situations. What counts in establishing the similarity of ***abc*** and ***mrrjjj*** is their shared successorship fabric. In ***mrrjjj***, that fabric has nothing to do with the specific letters *m*, *r*, and *j*; the letter sequence *m–r–j* is merely acting as a *medium* in which the *numerical* sequence "1–2–3" can be expressed. It is thus misguided

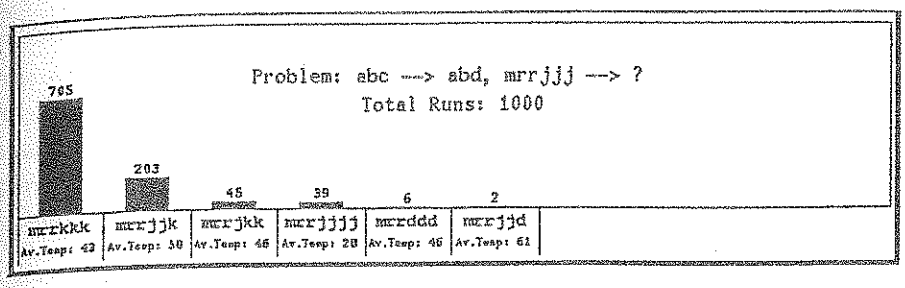

Figure V–4. Bar graph summarizing 1,000 runs of the Copycat program on the analogy problem "abc ⇒ abd; mrrjjj ⇒ ?".[4]

to focus on the *alphabetic* level of ***mrrjjj*** when one has just established that the essence of that string, in this context, is not its letters but its higher-level *numerical* structure. If the relations between lengths are perceived, then the answer at the level of lengths is 1–2–4. Translated back into the language of the carriers, this yields ***mrrjjjj***. Additionally converting the four *j*'s into *k*'s is gilding the lily: it simply blends the alphabetic view with the numerical view in an inappropriate manner.

People have also proposed answers such as ***mrryyyy***, where the three *j*'s are replaced by four copies of an arbitrary letter — here, *y*. The reasoning is that since the successorship fabric in ***mrrjjj*** has nothing to do with the specific letters ***m***, ***r***, and *j*, it doesn't matter *which* letter-value is used to replace the *j*'s. Such reasoning is too sophisticated for the current version of Copycat, which does not have the concept "arbitrary letter" — but even if Copycat could produce such an answer, we would still argue that ***mrrjjjj*** is the best answer to this problem. The letters ***m***, ***r***, and *j* serve as the medium for expressing the message "1–2–3", and we feel the most elegant solution is one that *preserves* that medium in expressing the modified message "1–2–4". Otherwise, why wouldn't an answer involving *total* replacement of the medium, such as ***uggyyyy***, be just as good as, if not better than, ***mrryyyy***?

As can be seen in Figure V–4, by far the most common answer is the straightforward ***mrrkkk***, with ***mrrjjk*** coming in a fairly distant second. For Copycat, these are the two most immediate answers; however, the average final temperatures associated with them are fairly high, because of the lack of any coherent structure tying together the target string as a whole.

Next come two answers with roughly equal frequencies: ***mrrjkk***, a rather silly answer that comes from grouping only the rightmost two *j*'s in ***mrrjjj*** and viewing this group as the object to be replaced; and ***mrrjjjj***. The average final

4. The differences between the frequencies given in this figure and those given for the same problem in Mitchell & Hofstadter, 1990b are due to several improvements in the program — in particular, improvements in the way letter-groups and bridges are constructed.

temperature associated with this answer is much lower than that of the other answers, which shows that the program assesses it to be the most satisfying answer, though far from the most immediate. As in many aspects of real life, the immediacy or obviousness of a solution is by no means perfectly correlated with its quality. The other two answers produced in this series of runs, ***mrrddd*** and ***mrrjjd***, come from replacing either a letter or a group with ***d***'s, and are on the fringes.

In Problem 4, the successorship fabric is between group lengths rather than between letters, and is thus not immediately apparent. A simple variant on Problem 4 involves a successorship fabric both at the level of letters and at the level of lengths:

5. Suppose the letter-string ***abc*** were changed to ***abd***; how would you change the letter-string ***rssttt*** in "the same way"?

The strong pressures evoked in Problem 4 by the lack of any alphabetical fabric are missing in this variant, and the effect on Copycat can be seen in the bar graph given in Figure V–5: the program gave the *length* answer ***rssttttt*** only once in 1,000 runs (as contrasted with 39 instances of ***mrrjjjj*** in Problem 4). In Problem 5, the program is much more satisfied with the *letter-level* answer (***rssuuu***), which dominates and has a relatively low average final temperature. The other answers are similar to the answers given in the previous problem (plus there are a few additional answers based on strange groupings of the target string).[5]

Paradigm shifts in a microworld

A different set of issues comes up in the following problem:

6. Suppose the letter-string ***abc*** were changed to ***abd***; how would you change the letter-string ***xyz*** in "the same way"?

Naturally, the focus is on the letter ***z***. One immediately feels challenged by its lack of successor — or more precisely, by the lack of a successor to *Platonic* ***z***, the abstract concept (as opposed to the instance thereof found inside the string ***xyz***). Many people, eager to *construct* a successor to Platonic ***z***, invoke the commonplace notion of circularity, thus conceiving of ***a*** as the successor of ***z***, much as January can be considered the successor of December, the digit '0' the successor of '9', an ace the successor of a king, or, in music, the note A the successor of G. This would yield ***xya***.

Invoking circularity in this way to deal with Problem 6 is a small type of creative leap, and not to be looked down upon. However, the general notion of

5. The current version of Copycat is not able to create two simultaneous bonds between two given objects (*e.g.*, both the *alphabetical* and *numerical* successorship bonds between ***r*** and ***ss***), so the program is at present unable to get what many people consider to be the best answer — namely, ***rssuuuu***.

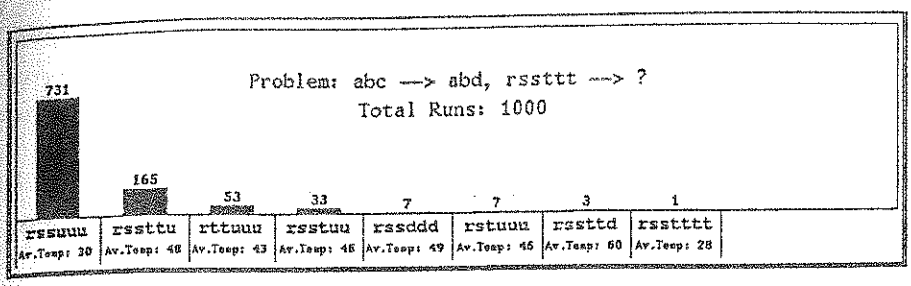

*Figure V–5. Bar graph summarizing 1,000 runs of the Copycat program on the analogy problem "**abc** ⇒ **abd**; **rssttt** ⇒ ?".*

circularity is not available to the program, as it is to people, for borrowing and insertion into the alphabet world. In fact, for important reasons, it is strictly stipulated that Copycat's alphabet is linear and just stops dead at *z*, immutably. We deliberately set this roadblock, because one of our main goals from the very conception of the project was to model the process whereby people deal with impasses.

In response to the *z*'s lack of successor, quite a variety of thoughts can and do occur to people, such as: replace the *z* by nothing at all, thus yielding the answer ***xy***. Or replace the *z* by the literal letter ***d***, yielding answer ***xyd***. (In other circumstances, such a resort to literality would be considered a rigid-minded and rather crude maneuver, but here it suddenly appears quite fluid and certainly reasonable.) Other answers, too, are possible, such as ***xyz*** itself (since the *z* cannot move farther along, just leave it alone); ***xyy*** (since you can't take the *successor* of the *z*, why not take its *predecessor*, which seems like second best?); ***xzz*** (since you can't take the successor of the *z* itself, why not take the successor of the letter sitting next to it?); and many others.

However, there is one particular way of looking at things that, to many people, seems like a genuine insight, whether or not they come up with it themselves. Essentially this is the idea that ***abc*** and ***xyz*** are "mirror images" of each other, each one being "wedged" against its own end of the alphabet. This would imply that the *z* in ***xyz*** corresponds not to the ***c*** but to the ***a*** in ***abc***, and that it is the ***x*** rather than the *z* that corresponds to the ***c***. (Of course, the ***b*** and the *y* are each other's counterparts as well.) Underlying these *object* correspondences (*i.e.*, bridges) is a set of three conceptually parallel slippages: *alphabetic-first* ⇒ *alphabetic-last*, *rightmost* ⇒ *leftmost*, and *successor* ⇒ *predecessor*. Under the profound conceptual reversal represented by these slippages, the raw rule flexes exactly as it did in Problem 2 — namely, into *replace the leftmost letter by its alphabetic predecessor*. This yields the answer ***wyz***, which many people (including the authors) consider elegant and superior to all the other answers proposed above. More than any other answer, it seems to result from *doing the same thing* to ***xyz*** as was done to ***abc***.

Note how similar and yet how different Problems 2 and 6 are. The key idea in both of them is to effect a double reversal (*i.e.*, to reverse one's perception of the target string both spatially and alphabetically). However, it seems considerably easier for people to come up with this insight in Problem 2, even though in that problem there is no "snag", as there is in Problem 6, serving to *force* a search for radical ideas. The very same insight is harder to come by in Problem 6 because the cues are subtler; the resemblance between ***a*** and ***z*** lurks far beneath the surface, whereas the resemblance between ***aa*** and ***kk*** is quite immediate.

In a sense, answer ***wyz*** to Problem 6 seems like a miniature "conceptual revolution" or "paradigm shift" (Kuhn, 1970), whereas answer ***hjkk*** to Problem 2 seems elegant but not nearly as radical. Any model of mental fluidity and creativity must faithfully reflect this notion of distinct "levels of subtlety". We will return to these issues of snags, cues, radical perceptual shifts, and levels of subtlety in the next main section, where we discuss the way in which Copycat succeeds, at least occasionally, in carrying out this miniature paradigm shift.

As can be seen in Figure V–6, the most common answer by far is ***xyd***, for which the program decides that if it can't replace the rightmost letter by its *successor*, the next best thing is to replace it by a ***d***. This is also an answer that people frequently give when told the ***xya*** avenue is barred.

A distant second in frequency, but the answer with the lowest average final temperature, is ***wyz***, which, as was said above, is based on simultaneous spatial and alphabetic reversal in perception of the target string. This discrepancy between rank-order by obviousness and rank-order by quality is characteristic of problems where creative insight is needed. Clearly, brilliance will distinguish itself from mediocrity only in situations where deep ideas are elusive.

To bring about pressures that get the idea of the double reversal to bubble up, radical measures must be taken upon encountering the "*z*-snag" (the moment when the attempt to take the successor of *z* fails). These include sharply focusing attention upon the trouble spot, and raising the temperature from its rather low value just before the *z*-snag is hit to its maximum possible value of 100, opening up a far wider range of possible avenues of exploration. Only with the special combination of a sharp focus of attention and an unusually "broad-minded" attitude could ***wyz*** ever be found. (We will discuss all this in more detail in the next section.)

The next answer, ***yyz***, reflects a view that sees the two strings as mapping to each other in a crosswise fashion, but ignores their opposite alphabetic fabrics; thus, while it considers the *leftmost* letter as the proper one in ***xyz*** to be changed, it clings to the notion of replacing it by its *successor*, since the letter changed in ***abc*** was replaced by *its* successor. (It is Problem 6's analogue to the answer ***jjkk*** in Problem 2.) Although this view seems somewhat inconsistent, like a good