Order Number 9116256

Copycat: A computer model of high-level perception and conceptual slippage in analogy-making

Mitchell, Melanie, Ph.D.

The University of Michigan, 1990

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

COPYCAT:

A COMPUTER MODEL OF

HIGH-LEVEL PERCEPTION AND CONCEPTUAL SLIPPAGE

IN ANALOGY-MAKING

by

Melanie Mitchell

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer and Communication Sciences)
in The University of Michigan
1990

Doctoral Committee:

Professor Douglas R. Hofstadter, Co-Chair
Professor John H. Holland, Co-Chair
Professor Arthur W. Burks
Professor Keki B. Irani
Assistant Professor Steven L. Lytinen
Professor Edward E. Smith

ACKNOWLEDGMENTS

My advisor, Douglas Hofstadter, was the prime mover behind the Copycat project: he created the domain and formulated the program's architecture, and he has guided every aspect of my work. Any success this project has had is due primarily to his insight and original thinking about thought. It was his book, *Gödel, Escher, Bach*, that originally convinced me to go into this field, and he has been an inspiring teacher as well as a true friend and supporter throughout my years in graduate school. His influence will without doubt continue to be strong and pervasive in my future work in artificial intelligence and cognitive science.

Two other professors at the University of Michigan have inspired and influenced me in significant ways: John Holland has taught me a great deal about complex systems, adaptation, and how to build computer models, and Stephen Kaplan has strongly influenced how I look at cognition. Both have had lasting effects on my view of what it is to be a scientist.

The other members of my doctoral committee—Arthur Burks, Keki Irani, Steven Lytinen, and Edward Smith—have given me many important suggestions for my research and for this dissertation. In particular, Edward Smith has helped me to think about how to demonstrate the model's psychological validity. David Pisoni and John Logan of Indiana University also helped me to design and run some of the psychological experiments described in this dissertation.

Robert French has been my closest cohort in graduate school. He has provided invaluable assistance on the Copycat project, and has also been a continual source of friendship, intellectual companionship, and moral support.

This project has been assisted and influenced in many ways by the other past and present members of Douglas Hofstadter's research group, including David Chalmers, Gray Clossman, Daniel Defays, Liane Gabora (who wrote the statistics-gathering program for

Copycat), Greg Huber, Helga Keller, Kevin Kinnell, David Leake, Roy Leban, Alejandro López, Gary McGraw, David Rogers, Peter Suber, and Henry Velick. Other fellow past and present graduate students at Michigan and elsewhere, including Jonathan Amsterdam, Lashon Booker, Stephanie Forrest, James Levenick, Wayne Loofbourrow, Rick Riolo, and Mark Weaver, have helped me in various ways in thinking about the issues in this dissertation. In particular, James Levenick read most of the chapters and provided many valuable comments.

I am very grateful to my parents, Norma and Jack Mitchell, for their unending love and support, as well as for their weekly "pep talks", which helped enormously to keep up my morale while I was writing this dissertation. My aunt, Faith Dunne, has also been a constant source of good advice and loving prodding throughout this process.

Finally, I want to thank David Moser, to whom I am more grateful than I can express, in part for many discussions about this project and for many helpful comments on this dissertation, but more importantly for being my most patient listener, my most enthusiastic supporter, and my dearest friend and companion.

My research has been financially supported by a University of Michigan Regents' Fellowship, and by grants to Douglas Hofstadter's research group from the National Science Foundation (grant DCR 8410409); the University of Michigan; Mitchell Kapor, Ellen Poss, and the Lotus Development Corporation; Apple Computer, Inc.; and Indiana University.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF APPENDICES

Appendix

CHAPTER I

OVERVIEW, PURPOSE, AND GOALS OF THE COPYCAT PROJECT

> To have a command of metaphor...is the mark of genius; for to coin good metaphors involves an insight into the resemblances between objects that are superficially unlike.
>
> —Aristotle, *The Poetics* (Cooper, 1913, p. 76)

> A cautious man should above all be on his guard against resemblances; they are a very slippery sort of thing.
>
> —Plato, *The Sophist* (Cornford, 1935, p. 180)

1.1 Introduction

This dissertation describes a computer program, called "Copycat", that is an implementation of a number of ideas about the mental mechanisms underlying high-level perception, conceptual slippage, and analogy-making in humans. The Copycat project was originally conceived by Douglas Hofstadter (1984a, 1985a) as part of a continuing research program in cognitive science, whose long-term goal is to understand the mechanisms underlying what Hofstadter calls the "fluidity" of concepts: their overlapping and associative nature, their indistinct boundaries, their dynamic and graded (rather than static and all-or-nothing) relevance in a given situation, their flexibility as a function of context—in short, their *fluid* rather than rigid adaptability to different situations. Such fluidity is a hallmark of human thought and its source is not well understood.

Hofstadter and his research group have been investigating this fluidity of concepts for many years and in a number of different domains, including pattern recognition, analogies, counterfactuals, speech and action errors, humor, translation between languages, and the

creation and recognition of different styles in such domains as typefaces, music, and art (Hofstadter, 1987). What is striking is that some of the same or similar mental mechanisms seem to underlie these seemingly disparate mental activities; in particular, central to all of them is the phenomenon of "conceptual slippage", in which certain descriptions in some mental representation are not held fixed, but are allowed to "slip"—that is, to be replaced by conceptually related descriptions in response to various kinds of pressures present in the situation at hand. For example, when we try to determine who the First Lady of Great Britain is (Hofstadter, 1985a), the usual definition "wife of the president" won't work, since, for one thing, Great Britain has no president. Thus the usual description of "First Lady" cannot be applied literally (unless you want to rigidly assert that Great Britain has no First Lady since it has no president). The concept has to be treated liberally, allowing some slippages. For instance, you might feel that the "president" of Great Britain is the *prime minister*, Margaret Thatcher, and that her "wife" is actually her *husband*, Denis. Thus, given the pressure of certain differences between the United States and Great Britain, the concepts *president* and *wife* slip to *prime minister* and *husband* respectively; these different concepts play the same *roles* in their respective situations. (People have also suggested numerous other candidates for the British First Lady, including Queen Elizabeth, and even her husband Prince Philip.) This notion of fluidly exporting *roles* (such as "First Lady") from one situation to another is fundamental to the mental phenomena that we are attempting to model.

Underlying this entire research program is a belief in the ubiquity and centrality of conceptual slippage in all aspects of thought, from basic and ordinary acts of recognition and categorization to rare and seemingly mystical feats of insight and creativity. Thus we believe that it is extremely important for researchers in cognitive science to isolate and study this phenomenon; the Copycat project is one attempt to do so.

Two previous projects carried out by Hofstadter and his graduate students—"Jumbo" and "Seek-Whence"—investigated certain aspects of perception and conceptual slippage, but each had a number of limitations, which will be discussed later in this dissertation. The particular goal of the Copycat project is to further develop ideas from these projects by building a model of how perception interacts with concepts to engender appropriate—and sometimes creative—conceptual slippages in the realm of analogy-making, a realm in which the necessity of constructing fluid and adaptable mental representations is particularly apparent.

The Copycat program interprets and makes analogies between situations in an idealized microworld (involving letter-string analogy problems). The program's architecture brings together many ideas, some inspired by other attempts at modeling perception, some inspired by naturally-occurring self-organizing systems. These ideas (to be explained more fully later) include:

- A parallel and self-organizing approach to building perceptual descriptions via the interaction of large numbers of independent "perceptual agents", with no global executive controlling the system's processing (inspired in part by the self-organizing mechanisms of metabolic processes in living cells and by the Hearsay-II speech-understanding program, Erman et al., 1980).

- A model of the concepts in which the composition of concepts, in terms of what conceptual slippages can be made, is not explicitly defined but rather emerges in response to what is perceived in the situation at hand. In this model, concepts attain various levels of activation in response to what is perceived, resulting in *shaded*—rather than black-and-white—levels of "presence" or "relevance" of various concepts in the situation at hand. Activated concepts spread activation to conceptual neighbors, and a concept's conceptual proximity to other concepts is dynamic and context-sensitive (changing according to current perceptions). Such a model has aspects in common with certain types of semantic networks, since concepts are modeled by nodes and links in a network, as well as with connectionist networks, since the degree of activation of nodes, the degree of association between nodes, and the constitution of concepts themselves are emergent outcomes of the interaction of the network as a whole with what is being perceived in the environment.

- An interaction of bottom-up (environment-driven) and top-down (concept-driven) modes of constructing perceptual descriptions, and a gradual transition from dominance of a bottom-up mode to dominance of a top-down mode, as organizing themes emerge from what has already been perceived.

- A notion of a *parallel terraced scan*, in which many different avenues of interpreting situations are explored simultaneously, each being explored at a speed and to a depth proportional to moment-to-moment evaluations of its promise.

- The use of *computational temperature* as a feedback device to measure the amount

and quality of global organization; this measure is then used to control the degree of randomness with which decisions are made in the system. The effect is to speed up the exploration of more promising avenues with respect to less promising ones as more and more information is obtained about them.

- A notion of *statistically emergent* high-level behavior, in which the system's low-level activities (involving mutually competitive and supporting actions by large numbers of independent perceptual agents) are permeated with nondeterminism, but more deterministic high-level behavior (e.g., the composition of concepts, the parallel terraced scan, and the actual interpretations created by the program for various situations) emerges from the statistics of the low-level nondeterminism.

The structure and contents of this dissertation are summarized as follows.

In this chapter, I discuss the relationship between analogy-making and the more general mental processes that I refer to as *high-level perception* and *conceptual slippage.*

In Chapter 2, I describe an idealized microworld developed by Hofstadter (1984a, 1984b) involving letter-string analogy problems, in which some of the central features of high-level perception and analogy-making are isolated and idealized. It is in this microworld that the Copycat program makes sense of situations and creates analogies between situations. I discuss the relation of the letter-string analogy problems in the microworld to analogy-making in the real world, and answer some of the commonly raised objections to using such a microworld for developing and testing cognitive models such as Copycat. Finally, I discuss the specific goals and criteria for success of my dissertation project. In particular, I propose a set of five analogy problems in Copycat's microworld whose solution would demonstrate many of the general abilities that Copycat is meant to model.

Chapter 3 first describes the Jumbo program, a direct predecessor of Copycat that explored some of the same issues, and from which sprang many of the ideas for Copycat. Next, the architecture of Copycat is described. The description is divided into two parts: a section giving an overview of the entire program, and then several sections describing the program in more detail. The overview section should be sufficient to give a general idea of how the program works, and the more detailed sections can be skipped or skimmed by readers for whom this overview is sufficient.

Chapter 4 gives statistics concerning Copycat's (and people's) solutions to the five target problems (discussed in Chapter 2) and gives annotated screen dumps from runs of Copycat

on the five problems. The screen dumps demonstrate most of the features of the program, and provide a more vivid explanation of how the program works.

Chapter 5 gives statistics for Copycat's (and people's) solutions to 27 variants of the five target problems, illustrating how the program responds to different pressures in the different variants.

Chapter 6 gives a discussion of some problems with the program as it currently stands.

Chapter 7 gives the results of selected "lesions" of Copycat, in which various aspects of the program's architecture were altered or removed in order to analyze the roles played by those aspects in the program's behavior.

In Chapter 8, I compare Copycat with related research on psychological and computational models of analogy-making, with related artificial-intelligence architectures such as semantic networks and production systems, and with connectionist and classifier-system models of concepts and learning. I discuss where Copycat lies in the spectrum that runs from so-called *symbolic* models of intelligence, which process information serially and in which concepts are represented as explicit data structures in a Lisp-like language, to *sub-symbolic* models, such as connectionist networks, in which processing is highly parallel and in which concepts are implicit and distributed over units in a network.

Chapter 9 concludes this dissertation with a summary of its main points, with proposals for future work on the Copycat project, and with a discussion of the contributions of this project to research in cognitive science and artificial intelligence.

Appendix A presents and discusses a number of analogy problems from Copycat's microworld that are currently beyond the program's capabilities.

Appendix B lists and describes parameters and formulas used in Copycat.

Appendix C gives more detailed descriptions than are given in Chapter 3 of the various types of codelets used in Copycat (the term "codelet" will be defined in Chapter 3).

Appendix D gives the results of two experiments (in addition to the survey whose results are given in Chapters 4 and 5) involving people's responses to various letter-string analogy problems.

As the above summary indicates, readers who want only an overview of what the Copycat project is about and how the program works should read Chapters 1–2, the first two sections of Chapter 3 (skimming the rest of the chapter if desired), Chapter 4, and Chapter 5. Readers who also want to know the results and limitations of the model should in addition read Chapters 6, 7, and 9, as well as Appendix A, and people who want to compare Copycat

with related research should read Chapter 8. Those who want a more complete and detailed description of the program should read all of Chapter 3 (Appendices B and C provide even more details).

This dissertation does not include the source code for Copycat (the program was written in Sun Common Lisp and its graphics run under the SunView window system). I will be happy to provide the source code to anyone who would like to use it for peaceful research purposes.

1.2 High-level Perception, Conceptual Slippage, and Analogy-Making

How, when one is faced with a situation, does understanding come about in the mind? How are we guided by a multitude of initially unconnected and novel perceptions to a coherent and familiar mental representation of an object or situation, such as "coffee cup", "the letter 'A'", "French Baroque style", or "the Vietnam of Central America"? And how are such representations structured so that they are flexible, fluid, and thus adaptable to many different situations, rather than brittle, rigid, and inextensible? This dissertation is a part of a broader research program focused on investigating the mental mechanisms underlying such acts of high-level perception and conceptual fluidity. Here, "high-level perception" refers to the recognition of objects, situations, or events at various levels of abstraction higher than that of of syntactic sensations tied to particular sensory modalities; it is to be distinguished from modality-specific mechanisms such as those of low-level vision. Another term for it would be "abstract recognition", referring to the recognition mechanisms we use when, say, we read a newspaper article about officials performing secret acts, shredding documents, lying to Congress, etc., and characterize these events as "a coverup", or "another Watergate".

High-level perception is intimately tied up with concepts: it is the act of applying previously stored concepts to describe and chunk parts of a new situation in order to build up a coherent mental representation. (Here, the term "situation" can refer to something as concrete as a coffee cup, or something much more complex and abstract, such as a certain political or social event.) But since every situation is different, recognition is not a mere matter of rigidly applying pre-defined, static concepts to describe aspects of an uninterpreted situation. An essential part of the recognition process is a mutual accommodation of one's concepts and one's developing mental representation of the situation at hand, as in the First Lady of Great Britain example, where both the concept of "First Lady" and the mental

representation of Denis Thatcher had to be reshaped in order to fit each other.

The process of recognizing concrete and abstract situations is more general than what is often referred to as *categorization*. The term "categorization" often implies that the situation is assigned to a single, previously stored, easily verbalizable category (such as "coffee cup" or "coverup"). However, in general, the "category" of a situation is often difficult to verbalize, yet recognizable nonetheless. Situations that abstractly remind one of other situations are often very clear examples of this. For example, a friend told me about a flight he took from Pittsburgh to Detroit (a 45-minute trip), in which the plane had been kept on the ground in Pittsburgh for an hour and a half in order to wait for a delayed shipment of soft drinks to be brought on board. The wait for the soft drinks defeated their whole purpose, which is to make the trip less tedious for the passengers, to make time seem to go by faster on the trip. This story instantly reminded me of how the University of Michigan Physical Plant Department fixed a decorative fountain on campus that often overflowed: they installed a large flotation device (of the kind found in toilet tanks) in the fountain's pool, which stopped the flow of water when the water level got too high. This made the fountain resemble a huge toilet tank, which of course ncompletely defeated its purpose, which was to be aesthetically pleasing and thus make the campus look more beautiful. I spontaneously recognized that the airplane situation was in the same, quite abstract category as the fountain situation—something like "situations in which an action taken to remedy a problem actually defeats the main purpose of the thing affected by the problem". It is not an easy category to verbalize. (Several examples of such reminding experiences are discussed by Schank, 1982.)

This example illustrates the blurry line between what we call "categorization" and what we call "analogy-making". Was my reminding experience a feat of analogy-making or of categorization? There is no clear distinction between the two. When a child learns that the words "mouth" and "drink" apply to a huge number of different objects and situations, is this categorization or analogy-making? When we describe Nicaragua as "another Vietnam", the Iran–Contra scandal as "Reagan's Watergate", or Nicolae Ceausescu as "the Stalin of Romania", are we categorizing or making analogies? One makes an analogy when one perceives non-identical objects or situations as being "the same" at some abstract level. Analogy-making is thus intimately related to recognition and categorization, for the essence of recognizing a cup (or a face, or a wave equation, or a symphony in the style of Mozart) is perceiving it to be "the same" at some level as other instances of that category. Very

similar, if not identical, mental mechanisms seem to underlie analogy-making, recognition, and categorization.

Turner (1988) makes a similar point: "Deeply entrenched analogical connections we no longer find inventive. We regard them as straightforward category connections." (p. 4). He characterizes the difference between analogical and categorical connections among concepts as a difference in the "degree of entrenchment" in the conceptual system rather than a difference in kind. Holyoak (1984) also makes a related point when he hypothesizes that analogical thinking may underlie the acquisition of schemas (abstract categories).

High-level perception thus encompasses recognition, categorization, and analogy-making, and its central feature is the fluid application of one's existing concepts to new situations.

At this point, I want to make clearer what the focus of our research is as it relates to the distinction between "concepts" and "categories". In colloquial speech, as well as in more formal psychological discussions, the two terms are used nearly synonymously (for example, this is the case in Smith & Medin, 1981, and in Lakoff, 1987), though there seems to be a subtle difference. From my own observations, it seems to me that there *is* a subtle difference between the way the terms "concept" and "category" are used colloquially. Very roughly, concepts are verbalized as singular nouns or phrases ("dog", "next-door neighbor", "socialized medicine") whereas categories are described using plural nouns or phrases ("dogs", etc.). There does seem to be some psychological difference between categories and concepts: what people generally call "categories" seem to be more directly associated with particular instances, whereas what people call "concepts" seem somewhat more distanced from their instances. I would characterize this difference roughly as follows: the word "concept" refers to a *symbol* in the mind for a class of instances, or for a single instance, and the word "category" refers more directly to the class itself (though not usually to a single instance).[1] Consider, for example, the following phrases: "the concept of 'hair color'" and "the category 'hair colors'". My impression is that mention of the latter is more likely to provoke mental imagery of particular hair colors than is the former. Another example (Moser, 1988) is "The Phoenicians invented the alphabet" versus "Alphabets are useful tools"; it seems that the latter would be more likely than the former to conjure up images of specific alphabets.

---

[1] This is similar in flavor to the distinction made in mathematics and philosophy between the *intension* and *extension* of a set.

This analysis is based on informal observations and intuitions, and is meant to be taken in that spirit. My characterization of this distinction is not perfect, but I can say, roughly, that this dissertation (along with the other research by Hofstadter's group) focuses more on what I am calling "concepts" than what I am calling "categories". That is, we are not so much concerned with the kinds of issues that psychological research on *categorization* deals with, such as *prototypes*, *exemplars*, *graded structure*, and other issues dealing with the internal structure of individual categories (e.g., Rosch & Lloyd, 1978; Smith & Medin, 1981; Lakoff, 1987); we are more concerned with the dynamics of the activation and association of *concepts*, as *active symbols* in the brain (Hofstadter, 1979, Chapter 11; 1985d), and how such symbols are used with flexibility to describe and relate different situations. The nature of this focus will become clearer in Chapters 2 and 3 when the psychological issues addressed by this project are spelled out, and the computer model is described.[2]

The "First Lady" example discussed above illustrates very strikingly how people use concepts with a great deal of flexibility. The concept "First Lady" is ordinarily taken to mean "the wife of the President", and thus it is easy to find the counterpart of the American First Lady in, say, Mexico, where there currently is a married, male president. However, as was discussed above, exporting the First Lady concept to Great Britain requires some flexibility; it requires certain concepts (*president* and *wife*) to slip into related concepts rather than being rigidly fixed. A reader might protest at this point that it isn't necessary for concepts to slip if we simply generalize the original definition of "First Lady" to "spouse of the head of state". However, more examples make it clear that the concept "First Lady", like other real-world concepts, cannot be crammed into so small a space. Hofstadter (1985a) gives two other examples that elegantly demonstrate this: when Pierre Trudeau was prime minister of Canada, many people considered his *former* wife Margaret to be Canada's First Lady, and for a long time during Jean-Claude Duvalier's reign in Haiti, the title of First Lady belonged to Simone Duvalier, his *mother* and the wife of the late *former president* François Duvalier. How to generalize "First Lady" now? Does it mean "spouse or parent, present or former, of head of state, present or former"? Even if such a verbose and awkward

---

[2] Note that this distinction between concepts and categories is quite different from that made by some psychologists. For example, Barsalou (1988) defines "concept" as "simply a particular individual's *conception* of a category on a particular occasion" (p. 93). He uses the word "concept" to refer to a temporarily constructed representation in working memory. As will be seen later on, I will refer to such representations as "perceptual structures". When I use the word "concept", I am referring to more permanent, long-term memory structures.

description were plausible, other undeniable instances of First Ladies would force further amendments (e.g., in some Muslim countries, where the king has more than one wife, or in the Philippines, where many people would say that Corazon Aquino holds the titles of President and First Lady simultaneously, and so on).[3]

Another problem with attempting to abstract away any differences by using a generalized definition such as "spouse of the head of state" is that, by adopting such a generalization, you lose the sense that Denis Thatcher is not a *normal* First Lady; you lose the sense of tension that comes from the strong feeling that the role should be filled by a woman. Such a sense of tension is essential to assessing the quality or interest of an analogy. Also, important information is lost in a generalization (e.g., *spouse*) that is present in a slippage (e.g., *wife* slips to *husband*); as will be discussed later on, in order to complete or extend an analogy, very often one has to keep track of *how* certain concepts slipped.

These examples are related to Lakoff's (1987) discussion of what he calls "radial" categories (e.g., "mother"), and perhaps an analysis similar to those presented by Lakoff could be applied to the concept of "First Lady", in which the meaning of the concept comes from the interaction of several different "models", such as *wife of the president*, *most distinguished woman in a particular field* (Greta Garbo has been called the "First Lady of film", Ella Fitzgerald the "First Lady of Jazz"), and so on. But the main point in presenting this example is to illustrate how subtly flexible and "slippery" real-world concepts can be, and how hopeless it is to try to come up with a definition or rule that will cover all past, present, and future cases. The view underlying this research project is that the only way to flexibly understand or categorize new situations is via conceptual slippage and analogy. This brings out once again the close relation between categorization and analogy-making.

The "First Lady" examples illustrate a central idea in this research: a view of concepts in which each concept consists of a central region surrounded by a halo of associated concepts

---

[3] David Moser (personal communication) has pointed out that much of the trouble a few years ago between Nancy Reagan and Raisa Gorbachev may have been due to a bad analogy on the part of Americans—namely, that "Raisa Gorbachev is the First Lady of the Soviet Union". This view caused many people (in particular, Mrs. Reagan) to be offended when Mrs. Gorbachev did not act in the way they felt a First Lady should properly act. However, it seems that Mrs. Gorbachev did not see herself as playing the same role in the Soviet Union as the role Mrs. Reagan played in the United States, and thus their encounters were rife with misunderstandings. (As it turned out, Mrs. Reagan, in her memoir (1989), acknowledged that the analogy was imperfect, noting—though perhaps inaccurately—that "There isn't even a Russian word for 'First Lady'", p. 337-338.)

which are potential slippages (e.g., *husband* is in the halo of the concept *wife*, and in some situations, such as identifying the First Lady of Great Britain, the description *wife* can slip to the description *husband*; that is, *husband* in one situation can be seen as playing the same role as *wife* in the other situation).

Conceptual slippage is ubiquitous in thought, and in some aspects of thought it can be seen especially clearly. For example, our speech and actions are permeated with errors involving conceptual slips, such as word substitutions (e.g., "Please fix the window, uh, I mean *mirror*" or "Are my legs, uh, I mean my *tires* touching the curb?") and action errors (e.g., stopping the car and unbuckling one's watch instead of one's seat belt, or trying to look up the word "February" in the dictionary by turning to the letter 'B'—both February and 'B' are second in a series).[4] Slippages are also apparent in the counterfactual statements people constantly make. You accidentally drop the milk bottle onto the floor, breaking it, and think, "I wish I had dropped the orange-juice jug instead, since it's made of plastic." Concepts tend to slip to close neighbors: *milk bottle* slips to *orange-juice jug* rather than to, say, *tablecloth*—it is very unlikely that you would think "I wish I had dropped the tablecloth instead; *it* wouldn't have broken." But of course the closeness—as well as the *availability* of other concepts as potential slippages from a given concept—depends on the situation. What slips, and how, depend on the interaction of specific *pressures* on the perceiver of the situation. As will be seen, the Copycat program is an attempt to model this context-dependent nature of conceptual slippage, to show how pressures in specific situations interact with concepts to provoke appropriate slippages. (See Hofstadter, 1979, Chapter 19, and Kahneman & Miller, 1986, for discussions of slippage in counterfactual thinking. Hofstadter & Gabora, 1990, gives a discussion of slippage in humor. Also, the role of slippage in translation between languages is discussed by Hofstadter, 1982, French & Henry, 1988, and Moser, 1989.)

The Copycat project concerns analogy-making, which provides a particularly clear window on the ways in which conceptual slippages take place. Since analogy-making is all about perceiving resemblances between things that are different, an analogy puts pressure on concepts to slip into related concepts, as is seen in the analogies involving First Ladies.

---

[4] There has been much research in psychology and linguistics on error-making; see, for example, Fromkin (1980), and Norman (1981). A large collection of interesting speech and action errors is given and discussed in Hofstadter and Moser (1989). The examples given above come from these three references.

Different people use the word "analogy" to mean different things, but when the term is taken in a very broad sense, to include all types of similarity comparisons ("this object is like that object", or "this situation is like that situation"), one cannot overestimate its ubiquity and importance at all levels of thought, from the most common and mundane acts of categorization to the most rare and significant feats of creation and discovery. Many researchers (e.g., Gentner, 1983) would hesitate to label categorization as a kind of analogy-making, but as the previous discussion in this chapter has illustrated, it is somewhat hard to draw a line between the two. As Gentner (1983) has pointed out, there is a spectrum of types of similarity comparisons. Our point here is that the mechanisms underlying these various mental activities are, if not the same, then at least very closely related.

The following are some examples of analogy-making (or, in some cases, its close cousins) along a spectrum from the everyday and mundane to the rare and exalted (though not necessarily perfectly ordered).

- A child learns the difference between cups and glasses, and can use the two words to correctly identify different objects.
- A child learns to recognize cats, dogs, boys, girls, etc. in books as well as in real life.
- A person is consistently and easily able to recognize the letter 'A', in spite of the fact that it appears in a vast variety of different shapes and styles, both in professionally designed typefaces and in different people's handwriting. There is something about all these 'A's that is essentially *the same.*
- A person is consistently and easily able to recognize that all the letters in a certain typeface (say, Helvetica) are in the same style; there is something about the letters that is essentially *the same.*
- Jean says to Simon, "I call my parents once a week". Simon replies, "Me too", not meaning, of course, that he calls *Jean's* parents once a week, but that he calls his own parents.
- A woman says to her male colleague, "I've been working so hard lately, I haven't been able to get enough time to spend with my husband", and he replies, "Yeah, me neither." He doesn't mean that he has no time to spend with *her* husband, or with *his own husband*, or even with his own *wife*, but rather with his *girl friend.*

- People take the suffix from "alcoholic", and use it to create new concepts like "workaholic", "chocoholic", "sexaholic", and "shopaholic".

- An advertisement describes Perrier as "the Cadillac of bottled waters". A newspaper article describes teaching as "the Beirut of professions". An opinion piece describes Saddam Hussein as "the Noriega of the Middle East".

- Nicaragua (or El Salvador) is called "another Vietnam". Cambodia is called "Vietnam's Vietnam". The Iran-Contra affair is called "Reagan's Watergate", and newspapers even dub it "Contragate".

- President Ronald Reagan calls the Nicaraguan Contras "Freedom Fighters", and likens them to "our Founding Fathers" in the American Revolution.

- A newspaper article portrays Denis Thatcher as the "First Lady of Great Britain".

- A jury acquits a man accused of rape because they judged that the victim was wearing "provocative" clothes, and was "asking to be raped". The National Organization of Women protests, asserting that this judgment is like saying that a person wearing an expensive watch is "asking to be robbed".

- Britain and Argentina go to war over the Falklands (or *las Malvinas*), a set of small islands near the coast of Argentina, populated by British settlers. Greece sides with England, because of its own conflict with Turkey over Cyprus, an island near the coast of Turkey, the majority of whose population is ethnically Greek.

- A classical-music lover hears an unfamiliar piece on the radio and easily recognizes it as being by Mozart. An early-music enthusiast hears a piece for baroque orchestra and can easily identify which country the composer was from. A studio composer arranges the Beatles' rock-and-roll hit "Hey Jude" in "easy listening" style to be played on Muzak radio stations.

- The linguist Zhao Yuanren translates *Alice in Wonderland* into Chinese, adapting the puns and other wordplay so that they work smoothly in Chinese while retaining the essence of the English original.

- The physicist Hideki Yukawa attempts to explain the nuclear force using an analogy with the electromagnetic force. On this basis, he postulates a mediating particle for

the nuclear force similar to the photon, which mediates the electromagnetic force. However, the new particle would have to mediate conversion of uncharged particles (neutrons) into charged particles (protons), and vice versa, which photons could not do. So certain slippages have to be made from the electromagnetic force to the nuclear force:

non-converter (photon) ⇒ *converter* (new particle),

which implies

uncharged (photon) ⇒ *charged* (new particle),

which in turn implies

massless (photon) ⇒ *massive* (new particle),

which requires a slippage from one type of equation to another:

massless equation ⇒ *massive equation.*

Yukawa uses these slippages to predict properties of the hypothesized particle (now known as a *pion*), which is subsequently discovered, and the predicted properties are verified (Yukawa, 1973a, 1973b).

- Johann Sebastian Bach takes a simple aria and creates a set of thirty variations on it (the "Goldberg Variations"), each one quite different and complex, many involving constraints not present in the original aria, but each containing something of the essence of the original in either melodic or harmonic structure.

There are two points to be made here. First, in an important sense, all of these *count* as examples of analogy-making. All are illustrations, at different levels of impressiveness, of the fluid rather than rigid nature of concepts and perception, by which the *essence* of a situation (be it a cup, a printed 'A', a family situation, a profession, a political situation, a piece of music or literature, a scientific idea, or whatever) can be distilled and fluidly transported to a different situation. Thus, though these examples range over a wide spectrum, there are fundamental psychological issues common to all of them; they reflect in different ways the same set of mental abilities, those of high-level perception and conceptual fluidity that have

been sketched in this chapter. The Copycat project is an attempt to extract and isolate some of these common issues and abilities and to propose a set of ideas about the underlying mechanisms. This proposal takes the form of a computer program that can make analogies in a microworld that contains many of these issues in an idealized form. Chapter 2 gives a description of the microworld and a discussion of the general issues it contains.

The second point is that these examples give some indication of the ubiquity and range of analogy-making in human thought. The analogy-making capacity in humans is far more than a mere tool used in the context of problem-solving, or a servant to a "reasoning engine". It is a central mechanism of cognition; it pervades thought at all levels, both conscious and unconscious, and cannot be turned on and off at will. (This view of the centrality of analogy in thought is complemented by the work of Lakoff and Johnson (1980) and Lakoff (1987), who provide evidence, using a vast array of linguistic metaphors, to argue that we understand all abstract and complex concepts (e.g., "love") by analogies to more direct perceptual experiences.)

In this section, the relation between analogy-making and high-level perception has been discussed, and the role of conceptual slippage in various mental processes has been illustrated by a number of examples. The point of the Copycat project is to investigate and model how perception interacts with concepts and how fluid conceptual slippages come about in the process of interpreting and making analogies between situations. The next chapter describes some of the specific issues in high-level perception and analogy-making that the Copycat program is meant to address, and illustrates how those issues arise in the idealized microworld in which the program makes analogies.

CHAPTER II

HIGH-LEVEL PERCEPTION, CONCEPTUAL SLIPPAGE, AND ANALOGY-MAKING IN A MICROWORLD

2.1 Copycat's Microworld

The research methodology of the Copycat project has been to attempt to isolate many of the central issues in high-level perception and analogy-making, to strip them down to their essence, and to construct a computer model that deals with these issues in this idealized form. This methodology is similar to that used by physicists, who typically attempt to solve idealized versions of problems that nonetheless capture the essence of the original problem. In basic sciences, particularly in physics, using such a methodology is indispensable in order to gain insight into deep underlying principles, because phenomena in the real world are often too complex to approach directly. For the same reasons, we believe that this methodology is also indispensable for approaching problems in cognitive science: much insight about mental mechanisms can be gained by looking at problems in a more stripped-down form without involving the vast amounts of information and complications of the real world that would make the construction of models intractable.

Adopting this isolate-and-idealize strategy, Hofstadter (1984a, 1984b, 1985a) has developed an idealized microworld for studying many of the essential features of perception and analogy-making. The basic objects in this world are the 26 letters of the alphabet, and analogy problems are constructed out of strings of letters, as in the following problem:

1. **abc** ⇒ **abd**
   ijk ⇒ ?

That is, one is asked, given the change from the string **abc** to the string **abd**, to do "the same thing" to the string **ijk** (i.e., one is asked to be a "copycat"—to copy the initial change, but using the material of the target string—hence the name of the program). The strings here are supposed to represent idealized situations containing objects, relationships between objects, and events; in this way they serve as metaphors for more complex real-world situations. The *initial string* **abc** and the *target string* **ijk** are two frameworks, each with its own objects and relationships. The change of **abc** to the *modified string* **abd** highlights a fragment of the first framework, and the challenge is to find "the same fragment" in the second framework (the target string), and to highlight and modify it in "the same way". What has been highlighted, how it has been highlighted, and what "the same way" means in the second framework are all up to the analogy-maker (human or machine) to decide.

The knowledge available to an analogy-maker in this microworld is fairly limited. The 26 letters are known, but only as members of a platonic linear sequence; shapes of letters, sounds, words, and all other linguistic and graphic facts are unknown. The only relations explicitly known are predecessor and successor relations between immediate neighbors in the alphabet. Ordinal positions in the alphabet (e.g., the fact that *S* is the 19th letter) are not known. (A note on notation: italic capitals (e.g., *S*) denote the 26 abstract *categories* (or *types*) of the alphabet, and never appear in strings; boldface smalls (e.g., **a**, **b**, and **c**) denote *instances* (or *tokens*) of those categories, and appear only in strings.) *A* and *Z*, being alphabetic extremities, are salient landmarks of equal importance. The alphabet is not circular; that is, *A* has no predecessor and *Z* has no successor. The alphabet is known equally well backwards and forwards (the fact that *N* is the letter before *O* is as equally accessible as the fact that *O* is the letter after *N*). In addition, strings (such as **abc** or **kkjjii**) can be parsed equally well left to right or right to left. The analogy-maker can count, but is reluctant to count above 3 or so, and has a commonsense notion of grouping by sameness or by alphabetical adjacency (forwards or backwards equally easily).

As can be seen from the description above, the knowledge assumed for this microdomain is not only limited, but is also different from that of people with respect to letter-strings (people can count far above 3, people usually know the alphabet better forwards than backwards, English speakers read left to right, and so on). The idea here is not to construct a model of how people solve letter-string analogy problems *per se*, but rather to construct a *domain* that, though idealized, captures much of the essence of real-world analogy-making,

so it can be used in developing a more general model. Thus there is a balance to be made in constructing such a domain: we want people to be able to understand and solve the letter-string problems without needing too much instruction about the restrictions of the domain, but we also want to avoid having features in the domain that are specific to the letter-string problems themselves (e.g., a left-to-right bias in reading), but are extraneous to the real issues of high-level perception and analogy-making that we are investigating. In addition, we exclude complex mathematical knowledge because we are trying to get at subconscious recognition processes rather than highly conscious "expert" activities like mathematics (such as noticing that the distance from *A* to *E* is twice as large as the distance from *M* to *O*).

For Problem 1 given above, a reasonable description of the **abc ⇒ abd** change is "Replace the rightmost letter by its successor", and straightforward application of this rule to the target string **ijk** yields the commonsense answer **ijl**. A more literal description (almost never given by people) is "Replace the rightmost letter by a *D*", yielding answer **ijd**. (This answer seems so literal-minded that many people laugh when it is suggested to them.) Even more literal-minded answers are **ijk** ("Replace any *C* by a *D*", and since there are no instances of *C* in **ijk**, just leave the target string alone) and **abd** ("Replace any string by **abd**"). However, these answers are very rarely given by people. People have necessarily evolved to be very good at describing things at an appropriate level of generality (i.e., appropriate for the purposes of living in the world), and this ability in the real world carries over to the abstract letter-string domain. Even though—technically—there are no "right" and "wrong" answers in a domain so divorced from real-world concerns, people fairly consistently agree on a single answer or a small set of answers as being the best response(s) to a given problem. (The results of some surveys of people on these problems will be given later on.) People's mental mechanisms have evolved for perception and analogy-making in the real world, but these mechanisms are still in operation even when the domain is artificial. Thus artificial domains such as the letter-string domain can be used to study general mental mechanisms.

In Problem 1 above, the rule "Replace the rightmost letter by its successor", describing the initial change **abc ⇒ abd**, can be applied straightforwardly to the target string **ijk**. However, other problems are not so simple. For instance, consider the same initial change and an alternate target string:

2. **abc** ⇒ **abd**
iijjkk ⇒ ?

Here a straightforward, rigid application of the original rule would yield **iijjkl**, which ignores the strong similarity between **abc** and **iijjkk** when the latter is seen as consisting of three *groups* of letters rather than as six *letters*. If one perceives the role of *letter* in **abc** as played by *group* in **iijjkk**, then in making a mapping between **abc** and **iijjkk**, one is forced to let the concept *letter* slip into the similar concept *group*. The rule for changing the target string becomes "Replace the rightmost *group* by its successor", yielding answer **iijjll**.

Consider now the following variant:

3. **abc** ⇒ **abd**
kji ⇒ ?

Here a literal application of the original rule would yield **kjj**, which again ignores a more abstract similarity between **abc** and **kji**. An alternative some people prefer is **lji** ("Replace the *leftmost* letter by its successor"), which is based on seeing **abc** as a left-to-right string and **kji** as a right-to-left string (where each string increases alphabetically); here there is a slippage from the concept *right* to the concept *left*, which in turn gives rise to the slippage *rightmost* ⇒ *leftmost*. Another answer given by many people is **kjh** ("Replace the rightmost letter by its *predecessor*"), in which **abc** is seen as increasing and **kji** as decreasing (both viewed as moving rightwards), yielding a slippage from *successor* to *predecessor*.

Notice that the same arguments would apply for the problem "**abc** ⇒ **abd**, **kjih** ⇒ ?" in which initial string **abc** is of length 3 and the target string **kjih** is of length 4. In the microdomain, as in real-world analogy-making, it is not necessary for there to be a one-to-one mapping between the objects of the two situations; an analogy can be made in spite of the fact that some objects, like the **b** in **abc** and the **j** and **i** in **kjih** have no clear counterparts in the other situation (just as an analogy can be made between the "First Family" of the United States and that of Great Britain without having to find a British counterpart for the Bushes' family dog).

Still other kinds of slippages can be seen in the answers to the following three problems.

4. **abc** ⇒ **abd**
ace ⇒ ?

Applying the original rule literally yields answer **acf**, which doesn't take into account the "double successor" structure of **ace** (*C* is the double successor of *A*, etc.). If **ace** is seen as

similar to **abc** because **abc** is a "successor group" and **ace** is a "double-successor group", then the answer is **acg** ("Replace rightmost letter by its *double successor*").[1]

5. **abc** ⇒ **abd**
   mrrjjj ⇒ ?

The answer that people give most often is **mrrkkk**, but this doesn't take into account the abstract similarity between **abc** and **mrrjjj**: **abc** increases *alphabetically* while **mrrjjj** consists of groups whose lengths increase *numerically*. If this similarity is perceived, than the answer is **mrrjjjj**, which reflects the view that the role played by *letter* in **abc** is played by *group-length* in **mrrjjj**, and requires a slippage from one to the other ("Replace rightmost *group-length* by its successor"). Even though people don't often produce this answer, when given a choice, some (not all) feel that it is a better answer than **mrrkkk** (see Appendix D). People occasionally give the answer **mrrkkkk**, replacing both the group-length and the letter-category of the rightmost group by their successors. This answer confounds aspects of the two situations. The strings **abc** and **mrrjjj** are similar since they both are woven together with the "fabric" of successorship, but this similarity is abstract, since in one case the fabric is successor relations between letters, and in the other case it is successor relations between group lengths. It thus seems strange to insist on retaining the notion of letter-successorship in the group-length situation where it no longer applies. It is as if a translator decided to tell the story of *War and Peace* in the context of the American Civil War, but gave the (now American) characters the names "Natasha" and "Alexey", refusing to let this aspect of the original novel slip. An even stranger translation would leave the names in the original Cyrillic letters, which might correspond to the answer **mrrdddd**. (Hofstadter has given the term "frame blends" to such mixtures of flexible and rigid thinking, which seem to be extremely common in thought, as well as being at the root of much humor of various kinds; see Hofstadter & Gabora, 1990.)

6. **abc** ⇒ **abd**
   aababc ⇒ ?

Here it is hard to make sense of the target string, and most people answer **aababd**, applying the rule "Replace rightmost letter by successor" directly. But if **aababc** is parsed as **a–**

---

[1] As mentioned earlier, the only relations explicitly known to analogy-makers in this microworld are immediate successor and predecessor relations, so in order to arrive at the answer **ace**, an analogy-maker would have to create the concept *double successor* on the fly.

ab–abc, then a strong though abstract similarity to the initial string **abc** emerges, where the "rightmost letter" of **aababc** is the group **abc**, and its "successor" is **abcd**, yielding answer **aababcd**.

I hope that the preceding problems help to make the case that although the analogies in this microworld involve only a small number of concepts, some of them require considerable flexibility and insight. A particularly clear example of such an analogy is the following:

7. **abc** ⇒ **abd**
   xyz ⇒ ?

At first glance, this problem is essentially the same as Problem 1 above (with target string **ijk**), but there is a snag: *Z* has no successor. Most people answer **xya**, but in Copycat's microworld the alphabet is not circular. This answer is intentionally excluded in order to force an impasse that requires analogy-makers to restructure their initial view, to make conceptual slippages that were not initially considered, and hopefully to discover a different way of understanding the situation. One such way is to notice that **xyz** is "wedged" against the far end of the alphabet, and **abc** is similarly wedged against the beginning of the alphabet. Thus the **z** in **xyz** and the **a** in **abc** can be seen to correspond, and then one naturally feels that the **x** and the **c** correspond as well. Underlying these object correspondences is a set of slippages that are conceptually parallel: *alphabetic-first* ⇒ *alphabetic-last*, *right* ⇒ *left*, and *successor* ⇒ *predecessor*. Taken together, these slippages convert the original rule into a rule adapted to the target string **xyz**: "Replace the *leftmost* letter by its *predecessor*", which yields a surprising but strong answer: **wyz**.

The seven problems discussed above give some idea of Copycat's microworld, but they are only a small sample from a vast space of interesting analogy problems involving letter-strings (Chapter 5 and Appendix A contain additional sample problems). These problems capture something of the flavor of the First Lady examples given in the previous chapter, illustrating how analogy-making requires fluid rather than rigid concepts: the process of making an analogy between two situations puts pressure on concepts in one situation (e.g., *president* and *wife*, or *rightmost*, *letter*, and *successor*), forcing them to slip into associated concepts in the other situation.

The current version of the Copycat program can deal only with problems whose initial change involves a replacement of at most one letter, which is why all the examples given above use the initial change **abc** ⇒ **abd** (of course the *answer* can involve a change of more

than one letter, as in Problems 2, 5, and 6), but this is a limitation of the program as it now stands; in principle, the domain is much larger.

2.2 Abilities Required for High-Level Perception and Analogy-Making

It is important to emphasize once again that the goal of this project is not to model specifically how people solve these letter-string analogy problems (it is clear that the microworld involves only a very small fraction of what people know about letters and might use in solving these problems), but rather to propose and model mechanisms for high-level perception and analogy-making in general. A very broad characterization of analogy-making can be given as follows: analogy-making consists of distilling the *essence* of one situation and *adapting* it to fit another situation. The letter-string analogy problems were designed to isolate and make very clear some of the mental abilities that are required in this process of understanding situations and perceiving similarity between situations. These abilities include the following (which, though listed separately, are of course strongly interrelated):

- Mentally constructing a coherently structured whole out of initially unattached parts;
- Describing objects, relations, and events at the "appropriate" level of abstraction;
- Chunking certain elements of a situation while viewing others individually;
- Focusing on relevant aspects and ignoring irrelevant or superficial aspects of situations;
- Taking certain descriptions literally and letting others slip when perceiving correspondences between aspects of two situations;
- Exploring many plausible avenues of possible interpretations while avoiding a search through a combinatorial explosion of implausible possibilities.

How each of these arises in Copycat and in real-world situations is discussed below.

Mentally constructing a coherently structured whole out of initially unattached parts. This description is very broad, and could be given as a *definition* of "recognition", but the point is that it applies not just to modality-specific recognition processes such as interpreting visual scenes, recognizing faces, or comprehending utterances, but to more abstract kinds of recognition as well, such as the recognition of a *coverup* (as discussed earlier). The letter-string problems in Copycat's microworld are given to the program basically unlabeled: relationships and correspondences between letters are not given ahead of time, and it is

up to the program to take the initially unattached letters and to weave them together into meaningful groupings and correspondences. The other abilities listed below are necessary for doing this.

Describing objects, relations, and events at the "appropriate" level of abstraction. What is "appropriate" of course depends on the situation. People tend to agree on how abstract the description of a situation should be; that is, they agree on which things should be perceived in terms of the *roles* they play and which should be perceived more literally. For example, if we see the recent Iran–Contra affair as "another Watergate", we focus on Ronald Reagan in his *role* as "President" (and thus the counterpart of Richard Nixon) rather than more literally as "a man named Ronald Reagan". There is often competition between different possible descriptions: we could describe Oliver North as "a lieutenant colonel" (there were no lieutenant colonels playing significant roles in Watergate) or as "the one who shredded the documents", perhaps viewing *him* as the counterpart of Nixon (viewing the latter as "the one who erased the tapes"). Or we might view North as "the scapegoat, who was following orders from higher up", seeing him as the counterpart of the Watergate burglars. Situations in the real world contain many different facets, and there is always competition among the various ways of perceiving these facets. Sometimes literal descriptions will be appropriate. Washington D.C., for example, is literally the same in both Watergate and Contragate.

This tension between literal descriptions and abstract roles is very evident in Copycat's letter-string analogy problems. For example, in Problem 1 of the previous section ("**abc** ⇒ **abd**, **ijk** ⇒ ?"), should the **c** in **abc** be described literally as "a *C*" or more abstractly, in terms of its *role* in its string—namely, "the rightmost letter"? (There are of course other possible roles one could perceive the **c** as playing, such as "the third letter in the string", "the highest letter in the alphabetic sequence", "the successor of the **b**", and so on.) Likewise, should the **d** in **abd** be described as "a *D*", or should the successor relationship with respect to the **c** be perceived? The answers to these questions depend on the context. For the given problem, the descriptions "Replace the rightmost letter by its successor" or "Replace the highest letter in the sequence by the next letter in the sequence" seem most appropriate (and are almost always the ones given by people when they are asked to describe the change), since one wants to give a description of a given situation that can fairly easily be exported to other situations, though without being too abstract and thus losing too much information. But consider the following problem:

1*a*. **abc** ⇒ **abd**
xcg ⇒ **?**

Here it would seem more reasonable than in the original problem to describe the **c** in **abc** at "face value" (i.e., using the rule "Replace *C* by *D*") since there is an instance of *C* in the target string as well, and the target string lacks the successorship structure of the initial string. Such a view would yield the answer **xdg**, whereas the "Replace rightmost letter by successor" view would yield the answer **xch**.

An intermediate case is:

1*b*. **abc** ⇒ **abd**
abcd ⇒ **?**

There is a tantalizing instance of *C* in the target string, tempting us to answer **abdd**, but there is also a shared structure between **abc** and **abcd**, in that both are increasing sequences beginning with *A*. The latter view lobbies for the answer **abce**, which is usually preferred by people.

Other variants, such as

1*c*. **abc** ⇒ **abd**
cde ⇒ **?**

and

1*d*. **abc** ⇒ **abd**
cba ⇒ **?**

illustrate variations and gradations in these pressures. The point is that this central issue of perceiving roles versus literal descriptions and describing elements of situations at "appropriate" levels of abstraction can be captured to some extent in the letter-string domain, small and restricted as it is. Moreover, as can be seen from the preceding examples, this issue can be explored in great detail in the microworld by constructing *families* of analogy-problems, as in 1*a*–*d* given above, where each member of a family varies a certain pressure along a certain dimension. Copycat's behavior on several such families of problems (including problems 1*a*–*d*) will be described in Chapter 5.

Chunking certain elements of a situation while viewing others individually. The issue of chunking is fundamental to perception at all levels. For instance, visually recognizing a chair requires mental chunking and labeling of its various parts (e.g., seat, back, arms).

Aurally interpreting a spoken sentence requires mental chunking of phonemes, syllables, words, and so on. Similarly, in more abstract forms of perception, making sense of a situation and making analogies between situations requires determining which parts should be viewed (and perhaps mapped onto the other situation) together as single units (e.g., in Watergate, one might chunk Haldeman and Ehrlichman as a unit, and then map that unit to a North–Poindexter unit in Contragate, or one might perceive Congress as a single unit in both situations). The chunking issue arises frequently in the letter-string problems; for example, in Problem 2 ("**abc** ⇒ **abd**, **iijjkk** ⇒ ?") and, more complexly, in Problem 6 ("**abc** ⇒ **abd**, **aababc** ⇒ ?"), where discovering a useful parsing for the target string is rather difficult.

Focusing on relevant aspects and ignoring irrelevant or superficial aspects of situations. Any complex situation has a huge number of aspects that can possibly be perceived, only some of which are relevant to a useful understanding of it. The ability to figure out which features are important and which can be ignored is fundamental to perception and analogy-making. For example, when looking for the Watergate counterpart of Ronald Reagan, do we care that Reagan has a wife named "Nancy"? Or when asking who played the role of Fawn Hall, do we pay attention to the fact that her boss was in the Marines? Do we care who was on the Senate investigating committee? This issue also plays a fundamental role in analogy-making in the letter-string domain. For example, in "**abc** ⇒ **abd**, **kjih** ⇒ ?", do we care that there are three letters in **abc** and four letters in **kjih**? Does the **b** in **abc** have to correspond to anything in **kjih**? And is it important to take into account that the rightmost letter of **abc** is an instance of *C*? Elements of situations don't come pre-labeled with the "right" description attached. Likewise they don't come pre-labeled as being "important" or "relevant". The perceiver is required to use both *a priori* knowledge and what has already been perceived about a given situation to determine which aspects are important and essential and which are irrelevant and superficial.

A very important point must be made here. The phrase used in the paragraph above, "the ability to figure out which features are important and which can be ignored", misstates the issue somewhat, since the problem is not, by any means, merely one in which many possible aspects of the situation are set before you and you have to decide which should be chosen for use in creating an interpretation or analogy. The process of perception involves not only deciding which *clearly apparent* aspects of a situation should be ignored and which should be taken into account, but how aspects that were initially considered to be irrelevant,

or were not even considered to be part of the situation in the first place, *become* apparent and relevant in response to pressures that emerge as the understanding process is taking place. In other words, sometimes, given certain pressures, concepts come into play that you initially didn't even suspect were part of the situation in any way.

As a example of these ideas (involving a commonly experienced situation), suppose you invite your good friend Greg to dinner, and he doesn't show up on time. What do you do? At first, simple, standard explanations and actions come to mind: he was briefly delayed; he ran into traffic; he had trouble parking. But as half an hour passes, then an hour, then two, the explanations and actions you think of become more and more out of the ordinary. The following might come to mind: call his office (no answer); call his apartment (no answer); check your calendar to make sure the dinner date is tonight (it is); rack your brains trying to remember if he warned you he might be late (you have no such memory); call friends of his to see if they know where he is (they don't); call his parents in Philadelphia (they haven't heard from him in weeks); call the police (they suggest checking the hospital); call the hospital (he's not there); go to his apartment (he's not there); ask his neighbors if they've seen him lately (they last saw him this morning); drive along routes he would likely have taken (he's nowhere to be seen); buy a megaphone and call out his name as you drive along; call several airlines to see if he's on a plane leaving town tonight; turn on the TV to see if you can spot him sitting in the audience of his favorite talk show; and so on. Though the last few are outlandish, most of these thoughts did occur to my friends and me when we were in such a situation. The point is: as time goes by and pressure builds up, one's thoughts go farther and farther out on a limb. One considers things that one never would have considered initially, letting seemingly unquestionable aspects of the situation slip under mounting pressure (e.g., Did I dream that I invited him? Did we have a falling-out that I forgot about? Did he leave town and not tell me?).

Not only are certain concepts explicitly present in one's mental representation of a situation (you consciously believe that Greg was driving); there are also implicit associations with those concepts, most of which stay well below the level of awareness. Given Greg's lateness, the thought that he's driving might easily evoke an image of his having trouble parking (a strong association). However, it is less likely that, early on, you will imagine him in a car accident. This weaker association is potentially there, but will not be brought into the picture without pressure (he is quite late, it is dark outside, etc.). This illustrates a general point: far-out ideas (or even ideas slightly past one's defaults) cannot continually

occur to people for no good reason; 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 on your block every time you go out to make sure the name hasn't changed. You don't check under your car for a hidden bomb every time you want to drive it. Likewise, counterintuitive ideas in science come about only in response to strong pressures. For example, had the Michelson–Morley experiment come out the other way (i.e., had it proved there is an "ether") and had Einstein still proposed special relativity, with all its deeply counterintuitive notions, it would have been seen as just a fascinating crackpot theory, not a great scientific advance. Not only is pressure needed for one to bring in previously uninvolved concepts in trying to make sense of a situation, but the concepts brought in are related to the source of the pressure. (This is related to the discussion in the previous chapter concerning what kinds of slippages are made in counterfactual thinking. These ideas overlap with Kahneman & Miller's 1986 treatment of counterfactuals.)

In short, flexibility in thought requires the potential for unexpected concepts to be brought into one's understanding of a situation, but only in response to pressure. An *a priori* absolute exclusion of a whole class of concepts initially assumed to be irrelevant is too rigid; one might then be prevented from coming up with unexpected new ways of looking at things. On the other hand, limitations of space and time make it impossible for *all* one's concepts to be made equally available for use in forming mental representations. A premise of the model being proposed here is that the presence or absence of a concept in a situation is not black-and-white; rather, all one's concepts should have the *potential* to become relevant in any situation, but due to the necessity for cognitive economy, they can't all be made available all the time or to the same degree. Instead, one must somehow manage to keep seemingly irrelevant concepts pretty much in the background most of the time, without absolutely and irrevocably excluding them.

These issues of graded relevance and availability of concepts in different situations come up often in the letter-string domain. For example, consider Problem 5:

5. **abc** ⇒ **abd**
mrrjjj ⇒ ?

You want to make use of the the salient fact that **abc** is an alphabetically increasing sequence, but how? This internal "fabric" of **abc** is a very appealing and seemingly explanatory aspect of the string, but at first glance, no such fabric seems to weave **mrrjjj**

together. So either (like most people) you settle for **mrrkkk** (or possibly **mrrjjk**), or you look 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 more satisfying answer. As was discussed above, if you ignore the letters in **mrrjjj** and look 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 **abc** ⇒ **abd** can be adapted 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**. But bringing the nonstandard concept of *length* into the picture requires strong pressures. These pressures include: top-down pressure to perceive successor relations in **mrrjjj** once they have been noticed in **abc**; the fact that once groups of letters are perceived in **mrrjjj**, the notion of *length* becomes weakly active and lingers in the background; and the decreased resistance to bringing in nonstandard concepts as organizing notions after more standard ones have failed to yield progress in making sense of the situation at hand. Thus this problem demonstrates how a previously irrelevant, unnoticed aspect of a situation emerges as relevant in response to pressures. The next two chapters describe and illustrate the mechanisms we are proposing for such capabilities.

Taking certain descriptions literally and letting others slip when perceiving correspondences between aspects of two situations. As was shown by the First Lady examples given in the previous chapter, even when roles (such as "wife of the president") have been perceived in a situation (e.g., the United States), they can't always be exported smoothly to a new situation (e.g., Great Britain). Either the roles have to be abstracted further ("spouse of the head of state") or slippages have to occur (*president* ⇒ *prime minister*, *wife* ⇒ *husband*). The process of perceiving correspondences between situations involves fights among pressures to use descriptions literally, to make descriptions more abstract, and to let descriptions slip into related descriptions. Notice that there is a distinction between *ignoring* certain aspects of a situation, because they are deemed to be irrelevant, and letting one's descriptions of certain aspects *slip*, precisely because they *are* relevant, but don't apply *as is* to the new situation. The letter-string domain was designed primarily to focus on the question of how different pressures interact to trigger appropriate slippages. Problems 2–7 from the previous section illustrate several different translations of the same rule ("Replace the rightmost letter by its successor"), each involving slippages triggered by different pressures

that come up in the perception of the different target strings.

Exploring many plausible avenues of possible interpretations while avoiding a search through a combinatorial explosion of implausible possibilities. A serious problem in trying to model perception is how to deal with the combinatorial explosion of possibilities. Competition must exist, but since the number of possible ways of interpreting a situation and making correspondences between situations is so large, not all possibilities can be explored fully (or even at all). This potential combinatorial explosion exists even for the simple situations in the letter-string domain (for example, in Problem 6, there are quite a number of possible ways in which the string **aababc** could be structured and mapped onto the string **abc**), though, of course, to a much lesser degree than for real-world situations. In any case, since the goal of the Copycat project is to propose and test mechanisms for high-level perception and analogy-making *in general* rather than specifically in the letter-string domain, we must make sure that the program does not take advantage of the small size of the microworld. Instead, the program must, as people do, have ways of circumventing the necessity of exhaustive search of any kind. To do so, it is necessary to use information as it is obtained to narrow the exploration of possibilities. For example, in making a Watergate–Contragate analogy, if you decide that that the notion of "erasing tapes" in Watergate corresponds to "shredding documents" in Contragate, then this view should make a mapping between Rose Mary Woods (Nixon's personal secretary, who erased tapes) and Fawn Hall (North's secretary, who shredded documents) more worthy of consideration than a mapping between, say, Woods and Reagan's personal secretary. Likewise, when one is solving Problem 3 ("**abc** ⇒ **abd**, **kji** ⇒ ?"), if *successorship* has been identified as a seemingly useful notion in the initial string, there should be top-down pressure to consider it in the target string as well. And if you perceive the two groups as increasing alphabetically but in different spatial directions (and thus make the slippage *right* ⇒ *left*), then a mapping between **c** and **k** (with the slippage *rightmost* ⇒ *leftmost*), becomes much more compelling, and consideration of a **c**–**i** mapping less likely. The process of using information as it is obtained involves not only allowing what is noticed to activate and reshape existing concepts in a bottom-up manner, but also allowing existing concepts to direct perception in a top-down manner. This interaction of bottom-up and top-down modes of processing is an essential part of the Copycat program. It will be discussed and demonstrated in detail in the next two chapters.

The notion of competing pressures. The abilities discussed above all illustrate how analogy-making necessarily entails competition among the huge number of possible ways of interpreting a situation and seeing similarities between situations. As can be seen in the various examples given above, a fundamental notion here is that of competing *pressures* acting on the perceiver/analogy-maker. It is through the competition among and resolution of these pressures that a coherent analogy emerges. (The same could be said for recognition processes in general.) Certain pressures are always strong in any analogy: for example, there is always a pressure to map salient things (such as "the President") onto other salient things, identical things (and very similar things) onto each other, and a pressure to use abstract descriptions (or roles) rather than literal descriptions. Any interesting analogy is the result of interaction and competition among a set of possibly conflicting pressures. The Copycat program is a model of this interaction and competition, and the letter-string microworld provides an arena in which all these various pressures arise in particularly clear ways. Also, the pressures can be minutely varied by constructing families of analogies such as the family given in 1*a–d* above.

2.3 The Issue of Retrieval

In Copycat's domain, both analogs (the initial and target strings) are given ahead of time, and some part of the initial string (e.g., **abc**) is highlighted by the presence of the modified string (e.g., **abd**), in which something has changed. This is very different from the usual way in which people make analogies: they are confronted with a situation, and either that reminds them of another situation with which they make an analogy (e.g., my "without-beverages-flight/overflowing-fountain" analogy), or they construct a fictional situation that is analogous to the original situation, in order to understand or to make some point about the original situation (e.g., the National Organization of Women's analogy between "provocative" clothing and an expensive watch). This problem of how people are reminded of situations or construct hypothetical situations has to do with the question of how memories are stored and retrieved. The Copycat project does not deal with this question directly, although many of the issues it does deal with—e.g., categorization, perception of similarity, slippage, and competition among interacting pressures—are closely related to questions about memory and retrieval. A faith underlying this research is that, for the time being, the problem of how people understand and make analogies between *given* situations can be investigated separately from the problem of how people are reminded of one thing

by another, though many of the same issues will be investigated, and research on the former will thus yield insights useful to research on the latter, and vice versa. This faith and methodology is shared by most cognitive-science researchers working on models of analogy-making (e.g., Gentner, 1983, Holyoak & Thagard, 1989, and Kedar-Cabelli, 1988b, among others), though the process of memory retrieval in the context of analogy-making has been investigated by, among others, Gick and Holyoak (1983); Schank and Leake (1989); and Thagard, Holyoak, Nelson, and Gochfeld (in press).

2.4 Defense of the Microworld

The following are the most frequently-raised objections to the use of the letter-string analogy problems for the purpose of constructing a model of analogy-making.

1. The problems are too simple and have no relation to "real-world" analogy-making.
2. They are not real analogies, but more like the proportional analogies on standardized tests (such as the Scholastic Aptitude Test) or like sequence-extrapolation problems.
3. Each problem is purposeless and none has any use in real-world problem-solving, so it is impossible to decide among rival answers to any problem.

1. *The problems are too simple and have no relation to "real-world" analogy-making.* I hope the discussion in the previous section has (at least partially) demonstrated the relation between the letter-string problems and "real-world" analogy-making. Although the lack of real-world flavor to the letter-string analogies makes some people find this research unconvincing, this lack is in some ways an advantage, in that it is very clear exactly what knowledge the program does have, and people are less likely to be fooled into believing that the program has an understanding of complex real-world concepts when it doesn't or that the program's behavior is more intelligent than it really is. It is sometimes too easy to ascribe intelligence to a program based on its seeming ability to deal with concepts that we, as humans, know a lot about, but about which the program actually knows almost nothing. This has been a recurring problem in artificial-intelligence research. The point of a microdomain in cognitive science is to isolate a phenomenon (such as analogy-making), to strip it down to its bare bones, to get rid of its extraneous real-world trappings, while at the same time retaining its essence so that it can be investigated more clearly. More

discussion of the merits and disadvantages of using microworlds will be given in Chapter 8, after some other computer models of analogy-making are discussed.

2. *They are not real analogies, but more like the proportional analogies on standardized tests (such as the Scholastic Aptitude Test) or like sequence-extrapolation problems.* There is a large difference between the usual "proportional" analogy problems on standardized tests such as the SAT (e.g., "foot : shoe :: hand : ?") and Copycat's letter-string analogies. Copycat's problems could be stated in the same form (e.g., **abc : abd :: iijjkk : ?**), but unlike the three-word SAT analogies, each string has quite a lot of internal structure to it, with many possible correspondences between the parts first and third strings instead of just one global correspondence between two atomic entities (e.g., "foot" and "shoe"). The letter strings are more like multi-part situations than like single words. A better comparison would be with the geometric analogy problems of Evans (1968), to be discussed in Chapter 8. Copycat's task has also been compared to sequence extrapolation. Problems based on the initial change **abc** $\Rightarrow$ **abd** have the flavor of sequence extrapolation, but the program is by no means limited to solving such problems. But even the **abd** $\Rightarrow$ **abd** problems are quite different from typical sequence-extrapolation problems, most of which use mathematical formulas that have little to do with the kinds of perceptual processes we are investigating. Other computer models of pattern perception and sequence extrapolation in strings of letters have been constructed (e.g., Simon & Kotovsky, 1963, to be discussed in Chapter 8), but the patterns used have generally not explored the range of issues discussed in the previous section.

3. *Each problem is purposeless and none has any use in real-world problem-solving, so it is impossible to decide among rival answers to any problem.* This objection seems to me to have two parts: 1) Is it possible to give any answer at all to these letter-string analogy problems? and 2) How can we say that, for a given analogy problem in this domain, one answer any is better than another? The first part comes from the claim that, because there is no notion of "purpose" in these letter-string problems (i.e., the analogies are not being used for solving real-world problems), there are no grounds for giving any answer at all. This objection seems to me to be easily refuted by the fact that people quite readily give answers to the letter-string problems and often have very strong opinions about the merit of their answer versus other answers. Moreover, in daily life, people make countless "purposeless" analogies all the time, by virtue of the fact that the human mind is continually perceiving, categorizing, and noticing all kinds of concrete and abstract similarities. Several examples

of such ubiquitous analogies (such as the "me too" analogies) were given in the previous chapter. A conscious purpose, if there is one, is one pressure among many, and usually is one of the factors that serve to highlight certain aspects of the source analog (Burstein & Adelson, 1987; Kedar-Cabelli, 1988b). For example, Kedar-Cabelli discusses how one could decide which features of a ceramic mug should be taken into account when making an analogy with a styrofoam cup, given that the purpose is to determine whether or not the latter could be used to drink hot liquids. But an analogy could certainly be made between the mug and the cup (in the sense that they could be seen as essentially similar) without that specific purpose in mind, since both have features that are salient *a priori*, that one notices even in the absence of any conscious purpose. A conscious purpose, being part of an overall context (often a relatively important part) serves to enhance the relevance of certain features. In the letter-string domain, the change from the initial string to the modified string (e.g., **abc** ⇒ **abd**) plays a similar role in that it highlights certain aspects of the initial string and helps indicate what aspects of the strings to take into account when making an analogy (e.g., the spatial positions—such as *rightmost* and *leftmost*—of elements in the string).

This notion of a conscious *purpose* as a *sine qua non* for analogy-making comes, I believe, from a somewhat narrow view in which analogy-making is seen as a tool to be used in problem-solving, rather than as a ubiquitous and pervasive mode of thought that blends smoothly into recognition and categorization.

The second part of the objection (about the possibility of judging the relative merit of answers) can also be countered in the same way: people *do* have preferences when answering these problems; they see certain answers as strong and others as weak or even ridiculous. Of course, there is not always universal agreement on the single "right" answer to a given problem; although there are always a small number of answers that people will give to a problem, preference *within* that set depends on individual taste.

In the real world, the analogies people find compelling are ones that take into account the essential features of situations and that strip away superficial and irrelevant aspects. Often, the more hidden or deep the shared essence, the more compelling the analogy. What is "essential" in a situation often has a very definite meaning in the real world: it is what *must* be perceived in order to survive and succeed in one's environment. In an artificial domain such as that of the letter-string analogy-problems, there is no such objective way of determining which answers are good and which are bad, but even so, people can feel

that certain ways of looking at a given problem are better than other ways, because they are using perceptual mechanisms that evolved to deal with real situations in the real world. These mechanisms cannot be turned on and off even when the domain is seemingly artificial and content-free, and when our survival does not depend on our actions. (This fact is implicitly acknowledged by the general acceptance of abstract visual analogy problems for use on intelligence tests; most people agree that their solution has *something* to do with intelligence.) Thus, since analogy problems in this artificial letter-string domain have an "essence" to be perceived, people perceive it, and are able to base their answers on it.

2.5 Specific Goals of This Dissertation and Criter ia for Success

In Chapter 1, I discussed in broad terms the goals of the research program of which the Copycat project is part. In this section I will outline the specific goals of my dissertation project, which are naturally much more limited than the very broad goals discussed earlier.

2.5.1 General Issues in Determining Criteria for Success

My goals for the Copycat project are two-faceted: first, that the program act with intelligence (albeit of a limited kind in a limited domain), and second, that its internal architecture and external behavior make it a plausible model of the aspects of *human* intelligence that we are investigating. These two facets are, of course, not simple to separate. It could be argued (and I believe is in part true) that the more intelligently the program acts, the more plausible it is as a model of human intelligence. However, it is certainly possible for a program to act with considerable intelligence—in a limited domain—but for its intelligence to arise from internal mechanisms that are very different from those of the human mind. A salient example of this is the recent rise of chess-playing programs to grandmaster status. These programs play chess by searching through a huge number of possible moves from a given board configuration, many moves into the future, and then selecting the move that promises the best future outcome. Psychologists agree that human chess-experts do not play in this way; rather, they rely upon high-level pattern-spotting abilities to recognize certain abstract patterns on the board and they then move in ways appropriate to those patterns (deGroot, 1965). Chess-playing programs act intelligently, but their intelligence comes from a very different source than that of people. However, chess is a very limited domain, and it is not at all clear that a program whose intelligence was more wide-ranging, and that included very fundamental human abilities such as abstract recognition and analogy-making, could

operate on principles very different from those underlying human intelligence.

A major goal of the field of artificial intelligence is to discover the general principles of intelligence, not necessarily its specific instantiation in *brains* (for instance, some AI researchers gain inspiration from other natural systems such as natural selection, e.g., Holland, 1986, or the immune system, e.g., Farmer, Packard, & Perelson, 1986). The goal of psychology, on the other hand, is to understand the mechanisms of human (or animal) intelligence and behavior. Thus the Copycat project is part artificial intelligence, part psychology. One motivation is the desire to understand in general how flexibility and adaptability—the hallmarks of intelligence—come about in complex systems (and, as will be discussed further in the next chapter, metaphors from both biology and society have been used in designing Copycat's architecture). Another motivation is the desire to understand the nature of fluid perception and concepts specifically in humans. These two very long-term goals are intimately related; it may be that one cannot be accomplished without the other. The hope is that Copycat not only acts intelligently, but it does so *because* it uses mechanisms like those of human intelligence, and thus more directly sheds some light on what these mechanisms are. Copycat as it currently stands is, of course, a far cry from human intelligence, even in its very limited domain. But the hope is that, in spite of its limitations, it captures something significant about the mechanisms of human perception and analogy-making, and that even where it is wrong it captures enough for it to be *interestingly* and *usefully* wrong. The hope is that the mechanisms being proposed have enough truth to them that the program's successes and failures say something interesting and helpful about what is right and wrong with these mechanisms.

The question to be answered in this section is, how are we to assess the program's success with regard to both its AI and psychological aspects? I want to show that Copycat acts with flexibility within its domain, that its concepts exhibit something like the fluidity and adaptability of human concepts (albeit over a very limited range of situations) and that its architecture and behavior have some psychological plausibility. Therefore, I am proposing two types of criteria for judging the program's success: first, *artificial-intelligence criteria*, which focus on the range of problems (and thus the range of issues in high-level perception and analogy-making) that the program can deal with, and second, *psychological criteria*, which focus on more specific comparisons of the program's behavior with that of people.

2.5.2 Artificial-Intelligence Criteria: What Problems the Program Can Deal With

The letter-string analogy-problem domain is very rich and open-ended; a very large number of issues in recognition and analogy-making can be explored in it. I think it is plausible to postulate that any computer program that could match human ability in this domain would be well along the way to being a generally intelligent program. (The current version of Copycat is, of course, very far from this.) For the purposes of this dissertation, I selected a set of five problems—Problems 1–3, 5, and 7, given in Section 2.1—as the main targets for the program. That is, the goal is for Copycat to solve these five problems, giving more or less the range of answers that people give (as will be seen in Chapter 3, the program is nondeterministic and can thus produce different answers on different runs).[2]

The point, of course, was not merely to solve a set of five problems, but to construct a program that is able to deal with the general issues that are contained (in an idealized form) in those problems. Each of the five problems requires different kinds of perceptual structures to be built and conceptual slippages to be made, so the fact that the program (whose mechanisms are meant to be general, not specific to the letter-string domain) can deal with these five cases demonstrates that its concepts do have a certain degree of fluidity in adapting to different situations.

Since the program uses general mechanisms to solve these problems, it can also solve a large number of other problems as well. The five target problems can be thought of as analogous to a basis in a vector space—each one defines a family (really, *multiple* families) of problems in which pressures are varied along different axes (as in problems 1*a–d* given earlier). In Chapter 5, Copycat's performance on 27 variants—family members—of the original five problems is displayed, demonstrating how robust and flexible the program is when it is stretched to deal with problems that it was not specifically designed to work on.

---

[2] The current version of Copycat can get the solution **aababcd** to "**abc ⇒ abd, aababc ⇒ ?**" in principle, but in practice it is too difficult for the program to discover and maintain the necessary parsing of the target string (**a-ab-abc**). Some of the weaknesses of the program that contribute to these difficulties will be discussed in Chapter 6. Copycat is currently unable to get the solution **acg** to "**abc ⇒ abd, ace ⇒ ?**" even in principle, since it lacks the ability to construct new temporary concepts such as *double successor*.

2.5.3 Psychological Criteria: More Detailed Comparisons With People's Behavior

The artificial-intelligence criteria described above are meant to test how intelligently Copycat acts, and to what degree its internal mechanisms yield a system with *fluid* concepts that can be used appropriately in a number of different situations. Gaining insight into the mechanisms underlying fluid concepts is the purpose of this project, so developing and testing a set of psychologically plausible mechanisms that produce such behavior is the main goal of the research described here. The range of the program's abilities and subjective judgments of the psychological plausibility of the proposed mechanisms can lend some credence to Copycat as a model of *human* psychological processes, but it is of course desirable, insofar as possible, to obtain further evidence that the mechanisms we are proposing are psychologically valid. The point of the psychological criteria is to see how well the program holds up under more detailed comparisons of its behavior with that of people.

There are some problems with designing and evaluating such comparisons, however. As was pointed out earlier, Copycat is not a model of how people solve letter-string analogy-problems *per se*, but rather, the letter-strings are meant to be taken as tiny abstract *models* of real-world situations. Copycat's knowledge of letters and strings is very limited, and doesn't involve many detailed aspects of human perception of letter-strings, such as the fact that English-speaking people read left to right, know the alphabet better forwards than backwards, and so on, since those aspects of perception are specific to letter-strings and are not relevant to the larger task of modeling recognition processes in general. Thus, since the program is not modeling the domain-specific aspects of how people solve letter-string analogy problems, direct comparisons between the details of how people solve these letter-string problems and how Copycat solves them (such as precise timing comparisons) are not useful for determining the program's psychological plausibility.

In spite of these difficulties, there are some comparisons that can be made. One reason the letter-string domain was chosen was because people *can* relate to it and can solve the problems; people generally have no trouble adapting to and obeying the restrictions of the domain as far as producing and judging answers are concerned. Thus there can be some useful comparisons between what people do and what the program does, as long as they are not at too fine-grained a level as far as the letter-string domain is concerned, or as far as the specific actions of the program are concerned. For the purposes of this dissertation, there are four types of comparisons that I chose to help further evaluate the program's

psychological plausibility. These are the following:

1. For a given problem, the program should be able (on different runs) to get all (or most) of the answers that people get (that is, people abiding by the constraints of the microworld), and should never produce answers that people find completely unjustified. Since the letter-string domain reflects general issues in high-level perception and analogy-making, the answers that people get are a function of how they respond to these general issues that are embedded in the letter-string problems. Thus the program's response to these general issues should be similar to that of people.

2. The program should respond to variations in pressures in a similar way to people. Problems 1*a–d* given above illustrated how pressures could be varied, and I discussed some of the effects these variations might have on what answers people tend to give; the goal here is for Copycat's tendencies to be affected in similar ways.

3. If people agree that there is a single obvious "best" answer to a problem (e.g., "**abc ⇒ abd, ijk ⇒ ijl**") the program should prefer that answer over the others it gets (as will be discussed in the next chapter, the program has a global variable called "temperature" whose value at the end of a run roughly indicates the program's "happiness" with the answer it produced).

4. The difficulties experienced by people should also be experienced by the program. People find some problems more difficult than others, so the program should experience roughly the same relative difficulties (provided that the difficulties are not due to something outside the domain, such as a case where one of the strings spells a word, etc.). For example, people universally find Problem 7 ("**abc ⇒ abd, xyz ⇒ ?**") harder than Problem 1 ("**abc ⇒ abd, ijk ⇒ ?**"); it would therefore be implausible if the program solved both with equal ease. One way to test this is to compare the relative times taken by people and by the program on these (and other) problems. Also, if people reliably experience a particular difficulty in solving a problem, the program should also experience that difficulty. For example, given "**abc ⇒ abd, xyz ⇒ ?**", if people always initially try to replace the **z** by its successor and hit an impasse, it would be implausible if the program were able to avoid this difficulty. Or if people often have a hard time making sense of the target string in "**abc ⇒ abd, mrrjjj ⇒ ?**", it would be implausible if the program easily noticed the relationships

among the group lengths. When I say that the program should experience the same difficulties as people, I don't mean that these difficulties should be "preprogrammed" in any sense, but that the behavior should emerge naturally from the mechanisms being proposed by the program's architecture, the mechanisms whose psychological plausibility is being evaluated.

Satisfying these criteria will not *prove* that the program has psychologically valid mechanisms; it will only show, to the degree that the criteria are satisfied, that the mechanisms it has are not implausible. And although these specific psychological tests can help to lend more plausibility to the model, the most important criteria are more general: Does the program exhibit flexible and insightful behavior in its microworld? Does it act like it has *fluid* concepts, as people do? Does it help us to better understand what concepts are?

CHAPTER III

THE ARCHITECTURE OF COPYCAT

In the previous chapter I discussed several general abilities that are required for high-level perception and analogy-making. The Copycat program is a model of the mental mechanisms underlying these abilities. In this chapter I describe the architecture of Copycat. The first section gives some background for the Copycat project by describing Jumbo, a computer program developed by Hofstadter in which some of the main ideas of Copycat's architecture were implemented in a limited and rough form. The second section gives a broad overview of how Copycat works, and the rest of the chapter gives more detailed descriptions of the various parts of the program's architecture. All but the first two sections can be skipped or skimmed by readers not desiring detailed knowledge of the program's workings.

3.1 Jumbo

Prior to the development of the Copycat project, Hofstadter originated two other computer-modeling projects—Jumbo and Seek-Whence—to investigate high-level perception and conceptual slippage. The Jumbo program was developed by Hofstadter (1983) and the Seek-Whence program by Meredith (1986). Jumbo was intended to be a short-term test of some ideas about high-level perception processes rather than a long-term project resulting in a sophisticated cognitive model. It was expressly designed as a "warm-up" for the more ambitious Seek-Whence and Copycat projects. Jumbo contained precursors of many of the features of Copycat, and it will thus be useful to briefly describe Jumbo's architecture before giving an overview of Copycat. (Seek-Whence will be discussed and compared with Copycat in Chapter 8.)

Jumbo's task was the creation of plausible anagrams (its name comes from the "Jumble"

anagram puzzles that commonly appear in newspapers): it was given a set of "jumbled" letters, and its job was to use those letters to create a single *English-like* word. Jumbo had no dictionary; its knowledge consisted of how, in English, consonant and vowel clusters are formed out of letters, syllables out of clusters, and words out of syllables. Given a set of isolated letters (atomic units), it was able to gradually and hierarchically construct "gloms" (chunks at the level of letters, clusters, syllables, or words).

The point of the program was to model, in a very simple domain, "the unconscious composition of coherent wholes out of scattered parts", the process of "constructing larger units out of smaller ones, with temporary structures at various levels and permanent mental categories trying to accommodate to each other" (Hofstadter, 1983). The program needed to engage in a large amount of back-and-forth notion in constructing, destroying, and regrouping structures; it needed data structures that were "fluidly reconformable" in the process of coming up with a single structure that included all the letters and obeyed some formal rules of English words—a "pseudo-word". Of course, all this could have been done more easily and quickly using a brute-force method in which all combinations were tried out and checked against a dictionary, but this would have defeated the purpose of the project, which was to construct a psychologically plausible model of some ideas for mechanisms underlying general perceptual processes.

The philosophy underlying Jumbo (as well as Seek-Whence and Copycat) is that high-level perception is not the result of using a set of serially applied, conscious mental rules, but rather, that it emerges as a statistical outcome of large numbers of independent activities occurring in parallel, competing with and supporting each other, and influencing each other by creating and destroying temporary perceptual constructs. Such a system has no global "executive" deciding which processes to run next and what each should do; rather, all processing is done locally by many simple, independent agents that make their decisions probabilistically. The system is self-organizing, with coherent and focused behavior being a statistically emergent property of the system as a whole. The presumption behind this philosophy is that the processes making up this "seething broth" of activity are below the level of consciousness, and thus cannot be examined introspectively, but that any computer model attaining a good degree of human-like flexibility will have to be implemented at this "subcognitive" level. These ideas have been discussed in detail by Hofstadter (1984a, 1985d); another argument for modeling at the subcognitive level is given by Smolensky (1988).

The architecture of Jumbo (as well as that of Seek-Whence and Copycat) was inspired in part by the Hearsay-II speech-recognition program. Hearsay-II's input is a raw, unperceived waveform, and the program consists of a number of independent "knowledge sources" that interact both cooperatively and competitively to hierarchically build up a coherent interpretation of the utterance. Likewise, in Jumbo all processing is done by "codelets": small, special-purpose pieces of code that act independently to build up hierarchical gloms from an initially unconnected set of jumbled letters.

In addition to the ideas from Hearsay-II, there were two metaphors that guided the development of Jumbo's architecture. The first metaphor involves biological cells—in particular, the way in which complex molecules are constructed in parallel and asynchronously by independent processes taking place throughout a cell's cytoplasm. For each type of molecule, there is a standard chemical pathway for assembling it, which may involve dozens of steps. The cell has no central executive coordinating these steps, but rather relies on more-or-less random encounters between enzymes and substrates for these construction activities to be carried out. The construction of complex molecules comes about as a result of wave after wave of enzymatic activity, in which products of one set of enzymes become substrates for the next wave of enzymes, and in which enzymes are themselves produced in response to the current "needs" of the cell. In Jumbo, codelets play the role of enzymes, randomly encountering letters and gloms (molecules) in the program's "cytoplasm" and attempting to join them to form ever-larger structures. Complex structures are built by *chains* of codelets.

For any given set of letters, there are, of course, many possible "glomming" paths to explore, just as in real-world perception, where there are a huge number of possible ways in which a set of unconnected raw sensations can be put together to form a global semantic interpretation. One of the purposes of Jumbo was to test out ideas about a strategy for efficiently searching though this potential combinatorial explosion of possibilities and quickly zeroing in on a good and coherent interpretation (or "pseudo-word"). These ideas were inspired in part by a second metaphor: the parallel, probabilistic, and dynamically self-adjusting search strategy used by people in looking for a mate. When searching for romance, you initially consider many people simultaneously—the people you happen by chance to meet (though this not totally random, since you tend to look in places where you think it will be more likely to meet interesting people). Some people you can dismiss almost immediately as possible romantic partners; they're clearly just not your type for some reason

or other. Others you consider a bit more seriously, though the amount of consideration is not equal; the amount of time and interest put into each possibility depends on your initial attraction to the person. You get to know some of these people better, and on the basis of your further evaluation of them (and also on the basis of how much they like you), you decide whom to concentrate on. You spend more and more time with a smaller and smaller set of people, though all the while perhaps still giving some small amount of consideration to one or two people you initially didn't find very interesting, or to new people that you happen to meet. Thus you are constantly reallocating your time and interest among the possible candidates for your affections according to how promising each relationship seems, until finally a relationship seems so promising that you decide to "commit", and give the lion's share of your time to this one person. However, commitment isn't necessarily the end of the story, since even after marriage, you are likely to meet other people, and might engage in "harmless flirtations" that, depending on their attractiveness and the strength of your commitment, might receive some amount of further consideration. And, depending on its seeming promise and on the state of your marriage, the rival exploration might even come to threaten your original commitment.

Hofstadter termed this type of strategy a *parallel terraced scan*: many possible courses of action are explored in parallel, but not all are are given the same amount of consideration or explored at the same speed. (the exploration is "terraced" because it is carried out in stages of increasing depth, with entry into each new stage being contingent upon the success of the previous stage). Possible paths are explored at a speed and to a depth proportional to moment-to-moment evaluations of their promise: the speed of an exploration process is locally adjustable to reflect the current assessment of the promise of the path being explored. In most situations in the real world, there are too many possibilities to explore; given real-time pressures, it is impossible to check them all out fully, or even to give *some* time to all of them. Instead, there has to be a parallel investigation of many possibilities to different levels of depth. The system can afford lots of quick forays, even into unlikely territory, but it cannot afford to explore all of them more deeply, much less to act upon every one. No path of exploration should be excluded *in principle*, though many have to be excluded *in fact*, since time is limited (e.g., when searching for a mate, you can't meet everyone in the world, though you probably shouldn't *absolutely* exclude anyone or any group *a priori*; who you *do* meet is probabilistic).

As in the romance example, as time goes on and progress is made, the mode of search

gradually changes from being highly parallel and random to being highly serial and deterministic. The Jumbo project proposed this strategy as a general feature of intelligence.

In the following description of glom construction (or destruction, or regrouping) in Jumbo, the influences from the cell metaphor and the romance metaphor on the program's architecture can be clearly seen. In Jumbo, a glom (such as "sch", say, from the set of letters "c o s h n o g i l") is built only after a series of evaluations of it are performed. The initial evaluations are quick and superficial, but later ones are more elaborate. If at any time one of these evaluations is too negative (decided probabilistically), the process of evaluation is curtailed. The glom is built only if the evaluation process goes all the way to the end. Each evaluation step is carried out by a codelet, and if the codelet's verdict is favorable, that codelet posts a new codelet to carry out the next evaluation in the series. Thus any given glom in Jumbo is built up by a standard series of codelet actions. An evaluation (made by a codelet) of a potential glom not only helps determine *whether* the evaluation process should continue, but also returns a numerical score, reflecting the codelet's estimate of *how promising* that particular glom is. That score is used to assign an *urgency* value to the next codelet in the series, which helps to decide how long that next codelet has to wait before it can run. (Similar multi-codelet chains also lead to the destruction or regrouping of gloms.) Attempts at building many different gloms are interleaved, as follows. All codelets waiting to run are placed in a data structure called the "Coderack", and at each time step, the system probabilistically chooses one codelet from the Coderack to run, the choice being based on the relative urgencies of all codelets in the Coderack at that time. When a codelet is chosen to run, it is removed from the Coderack. At the beginning of a run of the program, the Coderack contains a standard initial population of codelets, and as codelets run and are removed from the Coderack, they often add new, follow-up codelets to continue pursuing seemingly promising tasks. Like the enzyme population in a cell, the Coderack's population changes, as processing proceeds, in response to the needs of the system as judged by previously-run codelets.

Since all waiting codelets reside in the Coderack, and one codelet is chosen at a time from the entire Coderack population, a parallel terraced scan of possibilities results: in a given run, many competing or cooperating attempts at building gloms are interleaved, with the speed of each glom-building process being a statistical outcome of the urgencies of its component codelets. Thus many interleaved processes proceeded in parallel (though not in phase with each other), each at a speed and to a depth proportional to its estimated

promise, as assessed moment to moment.

In the Jumbo project, Hofstadter also first introduced his notion of a "temperature" variable, whose value reflects the overall "happiness" of the program at a given time—that is, how close the program estimates it is to creating a single coherent pseudo-word (the happier the program, the lower the temperature). This value in turn is used to control the amount of randomness with which decisions are made in the program. The temperature starts out high, and falls as structures are built. Since the program uses probability to choose which codelet runs next, the idea is that, as the program gets closer to a solution, the temperature falls, causing a decrease in randomness, which results in a speed-up of the rate at which promising possibilities are explored with respect to less promising ones. Thus good paths of exploration tend to crowd out worse ones at an ever-increasing rate, and the system is finally "frozen" into a solution when the temperature gets low enough. (The differences between this notion of temperature and that used in "simulated annealing" are discussed in the next section.)

In summary, the main ideas of the Jumbo program include 1) a cell-inspired architecture in which structures are built up in a piecemeal fashion by competing and cooperating chains of simple, independently acting agents (codelets), 2) a notion of fluid reconformability of structures built by the program (such as gloms), 3) a parallel terraced scan of possible courses of action, and 4) a temperature variable that dynamically adjusts the amount of randomness in response to how "happy" the program is with its currently built structures. The result is that the program's overall behavior is not directly programmed, but rather is a statistically emergent outcome of the interaction of many microscopic computational activities happening in parallel. All these ideas are the basis for Copycat's architecture as well. Since the point of the Jumbo project was to test out these ideas to some extent before using them in the Seek-Whence and Copycat projects, their implementation in Jumbo was by no means completely satisfactory. Many of the ideas, such as temperature, were only very sketchily implemented, and the program's sophistication was rather limited. Copycat is a much fuller and more sophisticated implementation of these basic ideas. It has required a great deal of further development and reworking of the basic ideas from Jumbo as well as the addition of many mechanisms not present in Jumbo.

3.2 Broad Overview of Copycat

Copycat's task is to use the concepts it possesses to build perceptual structures (descriptions of objects—i.e., letters or gloms, bonds between objects in the same string, groups of objects in a string, and correspondences between objects in different strings) on top of the three "raw", unprocessed strings given to it in each problem. The structures the program builds represent its understanding of the problem, and allow it to formulate a solution. Since for every problem the program starts out from exactly the same state with exactly the same set of concepts, its concepts have to be adaptable, in terms of their relevance and their associations with each other, to different situations. In a given problem, as the representation of a situation is constructed, associations arise and are considered in a probabilistic fashion according to a parallel terraced scan (as in Jumbo) in which many routes toward understanding the situation are tested in parallel, each at a rate and to a depth reflecting ongoing evaluations of its promise.

Copycat's solution of letter-string analogy problems involves the interaction of the following mechanisms:

- concepts consisting of a central region surrounded by a halo of potential associations and slippages, in which the relevance of the concept and the proximity to other concepts change as the process of perception and analogy-making proceeds;
- mechanisms for probabilistically bringing in concepts related to the current situation and conceptual slippages appropriate for creating an analogy;
- a mechanism by which concepts' relevances decay over time, unless reinforced;
- agents that continually seek new descriptions, bonds, groups, and correspondences in a working area;
- mechanisms for applying top-down pressures from concepts already deemed to be relevant;
- mechanisms allowing competition among pressures;
- the parallel terraced scan, allowing rival views to develop at different speeds;
- temperature, which measures the amount of perceptual organization in the system and, based on this value, controls the degree of randomness used in making decisions.

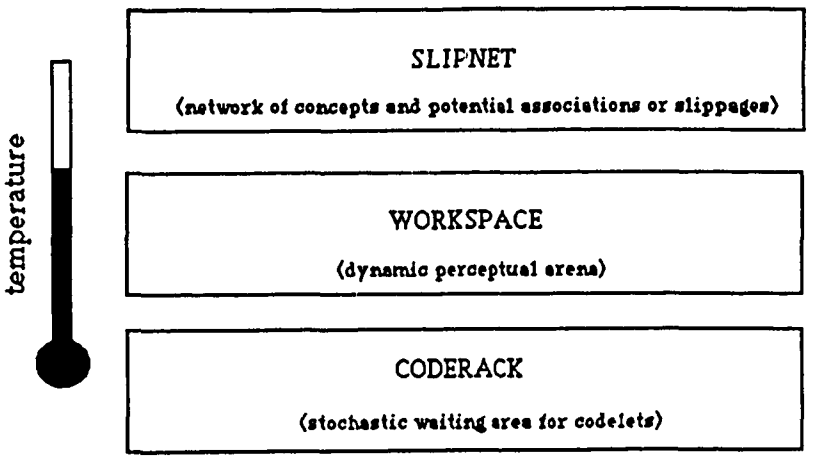


Figure 3.1: A schematic diagram of Copycat's architecture.

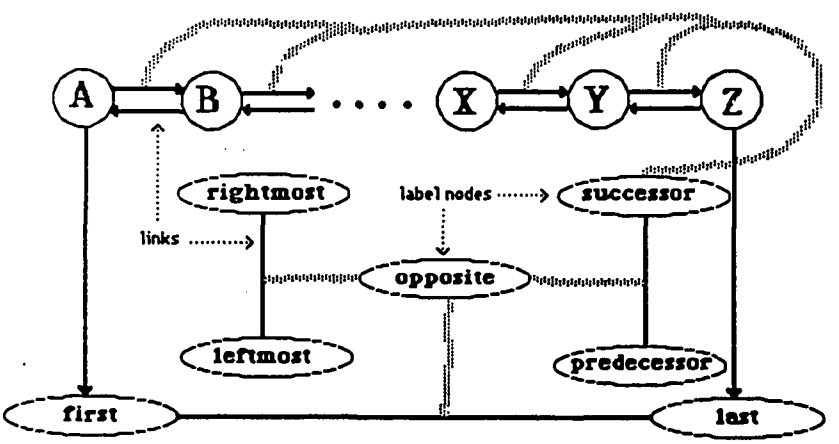


Figure 3.2: A small part of Copycat's Slipnet.

Figure 3.1 gives a schematic diagram of Copycat's architecture in which the four main elements are shown: the Slipnet, the Workspace, the Coderack, and a thermometer representing the temperature. An overview of each of these elements will be given in this section, and they will be described in more detail in the next section.

Copycat's concepts reside in á network of nodes and links called the *Slipnet* (so named because it is the source of all slippages). A small part of it is illustrated in Figure 3.2, which shows nodes, links (solid lines), and labels on links (thickly dotted lines).

A concept's central region is a node, and its associative halo corresponds to other nodes linked to the central node. A node (such as *successor* or *group*) becomes activated when instances of it are perceived (by codelets, as described below). Activation levels are not binary, but can vary continuously between 0 and 100% (at 100%, the node is said to be

"fully active"). The probability that a node will be brought in or be considered further at any given time as a possible organizing concept is a function of the node's current activation level. Thus there is no black-and-white answer to the question of whether a given concept is "present" at a given time; continuous activation levels and probabilities allow different concepts to be present to different degrees. All concepts have the potential to be brought in and used; which ones become relevant and to what degree depends on the situation the program is facing, as will be seen below.

A node spreads activation to nearby nodes as a function of their proximity; thus, conceptual neighbors of relevant nodes have the potential to be brought in as well. For example, the node *A*, when active, spreads some activation to *first* (since *A* is the first letter in the alphabet), giving the latter some probability of being used as a description). Nodes lose activation unless their instances continue to be perceived or to remain salient.

However, the rate of activation decay is not the same for all nodes. Each node has a preassigned *conceptual depth* value.[1] For example, the concept *A* is less deep than the concept *successor*, which is in turn less deep than the concept *opposite*. It could be said roughly that the conceptual depth of a node is the "distance" of an aspect of a situation from direct perception. For example, in the problem "**abc ⇒ abd, kji ⇒ ?**", the presence of instances of *A* is more directly perceived than the presence of *successorship*, which is in turn more directly perceived than the presence of *opposites*. Note that this conceptual-depth hierarchy is related to, but is not exactly the same kind of structure as an abstraction hierarchy such as *poodle–dog–mammal–animal–living-thing–thing*. The concept *A* could be said to be less abstract than the concept *successor*, which is in turn less abstract than the concept *opposite*, but, unlike descriptions on an abstraction hierarchy, these are not descriptions of the *same* object at different levels of abstraction. Rather, the idea is that different aspects of a given situation (e.g., the presence of an *A* or of *successorship* or of *opposites*) have different levels of depth as far as perception of them is concerned. Aspects with greater depth are more difficult to perceive, but tend to be more interesting and more useful in uncovering the essence of the situation. Once aspects of greater depth are perceived, they should have more influence on the ongoing perception of the situation than aspects of lesser depth. In Copycat, the greater a node's conceptual depth, the more slowly it decays, thus allowing deeper notions to persist longer (and thus have more influence on what structures are built)

---

[1] Hofstadter has called this measure "semanticity" (Hofstadter, 1984a).

than less deep ones. The conceptual-depth values in the Slipnet are fixed, and were pre-assigned by me based mainly on intuition (as well as trial and error). They are discussed in more detail in the next section and in Appendix B.

The length of a link between two nodes represents the conceptual proximity or degree of association between the nodes: the shorter the link, the greater the degree of association. (The apparent lengths of links in the diagram is not meant to represent their actual relative lengths.) Like activation, link-lengths are not constant, but can vary in response to what has been perceived. Many links have *labels* that are themselves nodes (e.g., the link between *rightmost* and *leftmost* is labeled by the node *opposite*). When a label node is fully active (indicating that the relationship it represents, e.g., *opposite*, is relevant to the problem at hand), all the links labeled by that node shrink—that is, such relationships are perceived as being closer, or more slippable.

Decisions about whether or not a slippage can be made from a given node—say, *rightmost*—to a neighboring node—say, *leftmost*—are made probabilistically, as a function of the conceptual proximity of the two nodes. (Such decisions are made by codelets, as is described later on.) For example, in the problem "**abc** ⇒ **abd**, **kji** ⇒ ?", if the program notices that the initial and target strings are alphabetically in opposite directions, then *opposite* will be activated, thereby increasing the probability of slippages between nodes connected by an *opposite* link such as *rightmost* ⇒ *leftmost*. Thus the plausibility of slippage between two nodes depends on context.

In this model, a *concept* (such as *rightmost*) is identified not with a single node but rather with a region in the Slipnet, centered on a particular node, and having blurry rather than sharp boundaries: neighboring nodes (such as *leftmost*) can be seen as being included in the concept probabilistically, as a function of their proximity to the central node of the concept. Just as in quantum mechanics, where the spatial position of an electron is "decided" only at the time it is measured, the composition of a concept in semantic space is decided only when slippages are explicitly made. For instance, in Problem 2 from Section 2.1 ("**abc** ⇒ **abd**, **iijjkk** ⇒ ?"), is the group **kk** an instance of the concept *letter*? If one makes a correspondence from the **c** to the group **kk**, then one is effectively saying, "in this context, yes"; that is what the slippage *letter* ⇒ *group* says. This is what we mean by "fluid concepts": people are able to take a rule like "Replace rightmost letter by its successor" and allow the words in it (such as "letter") to be flexibly extended. In this context, one sees the rule as "Replace rightmost 'letter' by successor", where the scare-quotes around

the word "letter" signify that here it is being used loosely to refer to a *group* as well. The whole point of the Copycat project is to study how perception and concepts interact so that one knows *which* words in the rule to allow to slip, and how to let them slip, while still maintaining the *essence* of the original rule. This model proposes that the property of *being a concept* is inextricably tied up with this notion of fluidity.

One could metaphorically compare a concept in the Slipnet to the metropolitan area of a city: for example, the New York metropolitan area is not a well-defined area with exact boundaries, but rather a central region diffusing continuously into outlying suburbs, with rather blurry edges. One could say that the conceptual proximity of a given location with New York is the probability that a person who lives there will answer "New York" when asked where they are from (note that for anyone from outside the strict city limits, this might entail a slippage, e.g., *Hoboken* ⇒ *New York*). The conceptual proximity here is context-dependent; it depends not only on the physical distance of the given location from the center of Manhattan, but also on who is asking the question and why, how familiar they are with New York, how interested they are in the answer, how much time there is to answer, how embarrassing it might be to answer "New Jersey", and so on. Likewise, in the Slipnet, the conceptual proximity from a given node and its neighbors is context-dependent; for example, in some situations (e.g., "**abc** ⇒ **abd**, **kji** ⇒ ?") the node *leftmost* may be closely associated with the node *rightmost*, making a slippage from one to the other more likely, whereas in other situations (e.g., "**abc** ⇒ **abd**, **ijk** ⇒ ?") the conceptual proximity and the likelihood of slippage is much less.

Since the conceptual proximity between two nodes is context-dependent, concepts in the Slipnet are emergent rather than explicitly defined. In other words, it is not preordained whether or not, say, *group* is part of the concept *letter* or *leftmost* part of the concept *rightmost*. As will be seen, the degree to which a given node is part of a given concept emerges from a large number of activities that take place as the program attempts to solve the problem it is faced with. Moreover, since the proximity between two nodes gives only the *probability* that a slippage will be possible, concepts are blurry, never explicitly defined. They are associative and dynamically overlapping (here, overlap is modeled by links), and their time-varying behavior (through dynamic activation and proximity) reflects the essential properties of the situations encountered. Thus concepts are able to adapt (in terms of relevance and association to one another) to different situations. Note that Copycat does not model *learning* in the usual sense: the program neither retains changes

in the network from run to run nor creates new permanent concepts. However, this project does concern learning if that term is taken to include this notion of adaptation of one's concepts to novel contexts.

In addition to the Slipnet, where long-term concepts reside, Copycat has a *Workspace*—in which perceptual structures are built hierarchically on top of the "raw" input (the three strings of letters). There are six types of structures that the program builds:

- *descriptions* of objects (e.g., *leftmost*);
- *bonds* representing relations (e.g., *successorship*) between objects in the same string;
- *groups* of objects in the same string (e.g, the **ii** group in **iijjkk**);
- *correspondences* between objects in different strings (e.g., a **c**–**kk** correspondence in "**abc** ⇒ **abd**, **iijjkk** ⇒ ?");
- a *rule* describing the change from the the initial to the modified string (e.g., "Replace rightmost letter by successor"); and
- a *translated rule* describing how the target string should be modified to produce an answer string (e.g., "Replace rightmost *group* by successor).

Copycat's Workspace is meant to correspond to the mental region in which representations of situations are constructed. (The counterpart of Copycat's Workspace in Jumbo was called the "Cytoplasm", reflecting the influence of the cell metaphor.) As in Jumbo, this construction process is carried out by large numbers of simple agents called *codelets*. A codelet is a piece of code that carries out some small, local task that is part of the process of building a structure (e.g., in Problem 5, one codelet might notice that the two **r**'s in **mrrjjj** are instances of the same letter; another codelet might estimate how well that proposed bond fits in with already-existing bonds; another codelet might build the bond). Bottom-up codelets (or "noticers") work toward building structures based on whatever they happen to find, without being prompted to look for instances of specific concepts; top-down codelets (or "seekers") look for instances of particular active nodes, such as *successor* or *sameness-group*. As in Jumbo, any structure is built by a series of codelets running in turn, each deciding probabilistically, on the basis of progressively deeper estimations of the structure's promise, whether to continue the evaluation process by generating one or more follow-up codelets or to abandon the effort at that point. If the decision is made

to continue, the running codelet assigns an urgency value (based on its estimation of the structure's promise) to each follow-up codelet. This value helps to determine how long each follow-up codelet will have to wait before it can run and continue the evaluation of that particular structure. Once a structure is built, it can indirectly influence the building of other structures, helping to accelerate the construction of structures that support it, and working to suppress the construction of structures that rival it. Incompatible structures cannot exist simultaneously; fights between such structures are decided probabilistically on the basis of strength. Which structures are incompatible, and how these supporting and competing actions take place will be described in subsequent sections.

Codelets can be viewed as *proxies for the pressures* in a given problem. Bottom-up codelets represent pressures present in all situations (the desire to make descriptions, to find relationships, to find correspondences, and so on). Top-down codelets represent pressures evoked by the situation at hand (e.g., the desire, in the problem "**abc** ⇒ **abd**, **mrrjjj** ⇒ ?", to construct more sameness groups in the target string once some have already been made).

Any run starts with a standard initial population of bottom-up codelets (with preset urgencies) on Copycat's *Coderack* (the place where posted codelets wait to be chosen); at each time step, one codelet is chosen to run and is removed from the current population on the Coderack. The choice is probabilistic, biased by relative urgencies in the current population. Copycat thus differs from an "agenda" system such as Hearsay-II, which, at each step, executes the waiting action with the highest estimated priority. The urgency of a codelet does not represent an estimated *priority*; rather, it represents the estimated relative speed at which the pressures represented by this codelet should be attended to. If the highest-urgency codelet were always chosen to run, the lower-urgency codelets would never be allowed to run, even though the pressures they represent have been judged to deserve *some* amount of attention. Using probabilities to choose codelets allows each pressure to get the amount of consideration it is judged to deserve, even when the judgments change as processing proceeds. This allocation of resources is an emergent statistical result rather than a preprogrammed deterministic one.

Codelets that take part in the process of building a structure send activation to the areas in the Slipnet that represent the concepts associated with that structure. These activations in turn affect the makeup of the codelet population, since active nodes (e.g., *successor*) are able to add codelets to the Coderack (e.g., top-down codelets that try to find successor relations between pairs of objects). Thus, as the run proceeds, new codelets are added to

the Coderack population either as follow-ups to previously-run codelets, or as top-down scouts for active nodes. (Also, new bottom-up codelets are continuously being added to the Coderack.) A new codelet's urgency is assigned by its creator as a function of the estimated promise of the task it is to work on: the urgency of a follow-up codelet is a function of the result of the evaluation done by the codelet that posted it and the urgency of a top-down codelet is a function of the activation of the node that posted it (the urgency of a bottom-up codelet is fixed). Thus the codelet population on the Coderack changes as the run proceeds, in response to the system's needs as judged by previously-run codelets and by activation patterns in the Slipnet, which themselves depend on what structures have been built.

The speed of a structure-building process emerges dynamically from the urgencies of its component codelets. Since those urgencies are determined by moment-to-moment estimates of the promise of the structure being built, the result is that structures of greater promise will tend to be built more quickly than less promising ones. The upshot is a parallel terraced scan—more promising views tend (statistically) to be explored faster than less promising ones. There is no top-level executive directing processing here; all processing is carried out by codelets. Note that though Copycat runs on a serial computer and thus only one codelet runs at a time, the system is roughly equivalent to one in which many independent activities are taking place in parallel, since codelets work locally and to a large degree independently.

The fine-grained breakup of structure-building processes thus serves two purposes: (1) it allows many such processes to be carried out in parallel, by having their components interleaved; and (2) it allows the computational resources allocated to each such process to be dynamically regulated by moment-to-moment estimates of the promise (reflected by codelet urgencies) of the pathway being followed.

It is important to understand that in this system, such processes, each of which consists of many codelets running in a series, are themselves *emergent* entities. Rather than being predetermined and then broken up into small components, processes are instead post-determined, being the pathways visible, after the fact, leading to some coherent macroscopic act of construction or destruction of perceptual or organizational structure. In other words, only the codelets themselves are predetermined; the macroscopic *processes* of the system are emergent. In short, any sequence of codelets that amounts to a coherent macroscopic act can *a posteriori* be labeled a process (e.g., the process of forming a successor group out of the entire string **iijjkk**, which involves the building of several bonds and the formation of several groups within the string), but large-scale processes are not laid out in advance.

For example, there is nothing in the program that says, "see if the target string can be made into a successor group" with instructions on how to do so; there are only individual codelets that perform small, local actions. Thus, though when one looks back at a run of the program and can identify certain "processes" that were interleaved and that ran at different speeds, it must be made clear that the way this actually comes about is very different from standard time-sharing systems in which a number of well-defined and separate processes are each given certain time-slices as a function of their priorities. At any given time during a run of Copycat, one cannot take the codelets currently in the Coderack and determine which ones belong to which large-scale process. Until the program runs to completion, it cannot be said what the various processes *are*; is even unclear what should be dubbed a process. Each codelet's task is one step in a very large number of potential processes that may or may not unfold. To describe the program's large-scale actions in terms of "processes", as has been done here, is really just a convenient shorthand.

A final mechanism, *temperature* (discussed earlier with respect to the Jumbo project), both measures the degree of perceptual disorganization in the system (its value at any moment being a function of the amount and quality of structure built so far) and controls the degree of randomness used in making decisions (e.g., which codelet should run next, which objects a codelet should choose to work on, which structure should win a fight, etc.). Higher temperatures reflect the fact that there is little information on which to base decisions; lower temperatures reflect the fact that there is greater certainty about the basis for decisions. Thus, decisions are made more randomly at higher temperatures than at lower temperatures. The final temperature at the end of a run can be taken as a rough indication of the program's satisfaction with the answer it has created (the lower the temperature, the better).

Note that the role of temperature in Copycat (and Jumbo) differs from that in simulated annealing, an optimization technique sometimes used in connectionist networks (Kirkpatrick, 1983, Hinton & Sejnowski, 1986, Smolensky, 1986). 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 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.

In summary, Copycat's temperature-controlled nondeterminism allows the program to

avoid an apparent paradox in perceiving situations: you can't explore every possibility, but you don't know which possibilities are worth exploring without first exploring them. It is necessary to carry out some exploration in order to assess the promise of various possibilities, and even to get a clearer sense of what the possibilities *are* (e.g., the notion of *length* in "**abc ⇒ abd, mrrjjj ⇒ ?**", which initially might not even be considered a possibility to explore). It is essential to be open-minded, but the territory is too vast to explore everything. In Copycat, the fact that codelets are chosen probabilistically rather than deterministically allows the exploration process to be a fair one, neither deterministically excluding any possibilities *a priori*, nor being forced to give equal consideration to *every* possibility. This is the role of nondeterminism in Copycat: it allows different pressures to be given the amount of consideration they seem to deserve, with this allocation of resources shifting dynamically as new information is obtained. There is much redundancy at the level of individual codelets, especially among codelets exploring the most promising possibilities, and the action of any one codelet does not make a difference in the program's overall behavior. Rather, all high-level effects, such as the parallel terraced scan, are statistical results of large numbers of codelet actions and probabilistic choices made by the program of which codelets to run. (A typical run of Copycat consists of hundreds—or sometimes, depending on the problem, thousands—of codelet steps.)

Copycat's distributed asynchronous parallelism, like Jumbo's, was inspired by the similar sort of self-organizing activity that takes place in a biological cell (Hofstadter, 1984a). As was outlined in the discussion of Jumbo, in a cell, all activity is carried out by large numbers of widely distributed enzymes of various sorts. These enzymes depend on random motion in the cell's cytoplasm in order to encounter substrates (relatively simple molecules such as amino acids) from which to build up larger structures (such as proteins). Complex structures are built up through long chains of enzymatic actions, and separate chains proceed independently and asynchronously in different spatial locations throughout the cytoplasm. Moreover, the enzyme population in the cell is itself regulated by the products of the enzymatic activity, and is thus sensitive to the moment-to-moment needs of the cell. In Copycat, as in Jumbo, codelets roughly act the part of enzymes. All activity is carried out by large numbers of codelets, which choose objects in a probabilistic, biased way for use in building structures. As in a cell, the processes by which complex structures are built are not explicitly programmed, but are emergent outcomes of chains of codelets working in asynchronous parallel throughout Copycat's Workspace (its "cytoplasm"). And just as

in a cell, the population of codelets on the Coderack is self-regulating and sensitive to the moment-to-moment needs of the system. To carry this analogy further, the Slipnet could be said to play the role of DNA, with active nodes in the Slipnet corresponding to genes currently being expressed in the cell, controlling the production of enzymes. Hofstadter's purpose in inventing this metaphor was to draw inspiration from the mechanisms of self-organization in a fairly well-understood natural system, and to use these ideas in thinking about the mechanisms of high-level perception. The mechanisms of enzymes and DNA in a cell are not to be taken literally as a model of perception; rather, general principles can be abstracted and carried over from the workings of cells to the workings of perception. Distributed asynchronous parallelism, emergent processes, the building-up of coherent complex structures from initially unconnected parts, self-organization, self-regulation, and sensitivity to the ongoing needs of the system are all central to our model of perception, and thinking about the workings of the cell has helped in devising mechanisms underlying these principles in Copycat.

This section has described the Copycat program in very broad strokes; in the rest of this chapter, the various parts of the architecture sketched above will be described in more detail.

3.3 The Slipnet

Figure 3.3 is an expanded version of Figure 3.2. All the nodes and links in the Slipnet are shown in this figure. Note that the sizes of nodes and the lengths of links in this diagram are arbitrary, and do not indicate anything about the actual nodes and links.

The network includes nodes representing the following possible descriptors for objects and structures.

- The 26 letters of the alphabet.
- The numbers 1 to 5 (the program doesn't know any numbers higher than this, and currently the only way the program uses numbers is to describe the lengths of groups).
- The various possible positions of an object in a string: *leftmost*, *rightmost*, and *middle*, and the nodes *whole*, which is used to describe a grouping of a whole string, and *single*, which is used to describe a letter that is the sole constituent of its string.
- The two possible spatial directions for bonds and groups: *left* and *right*.

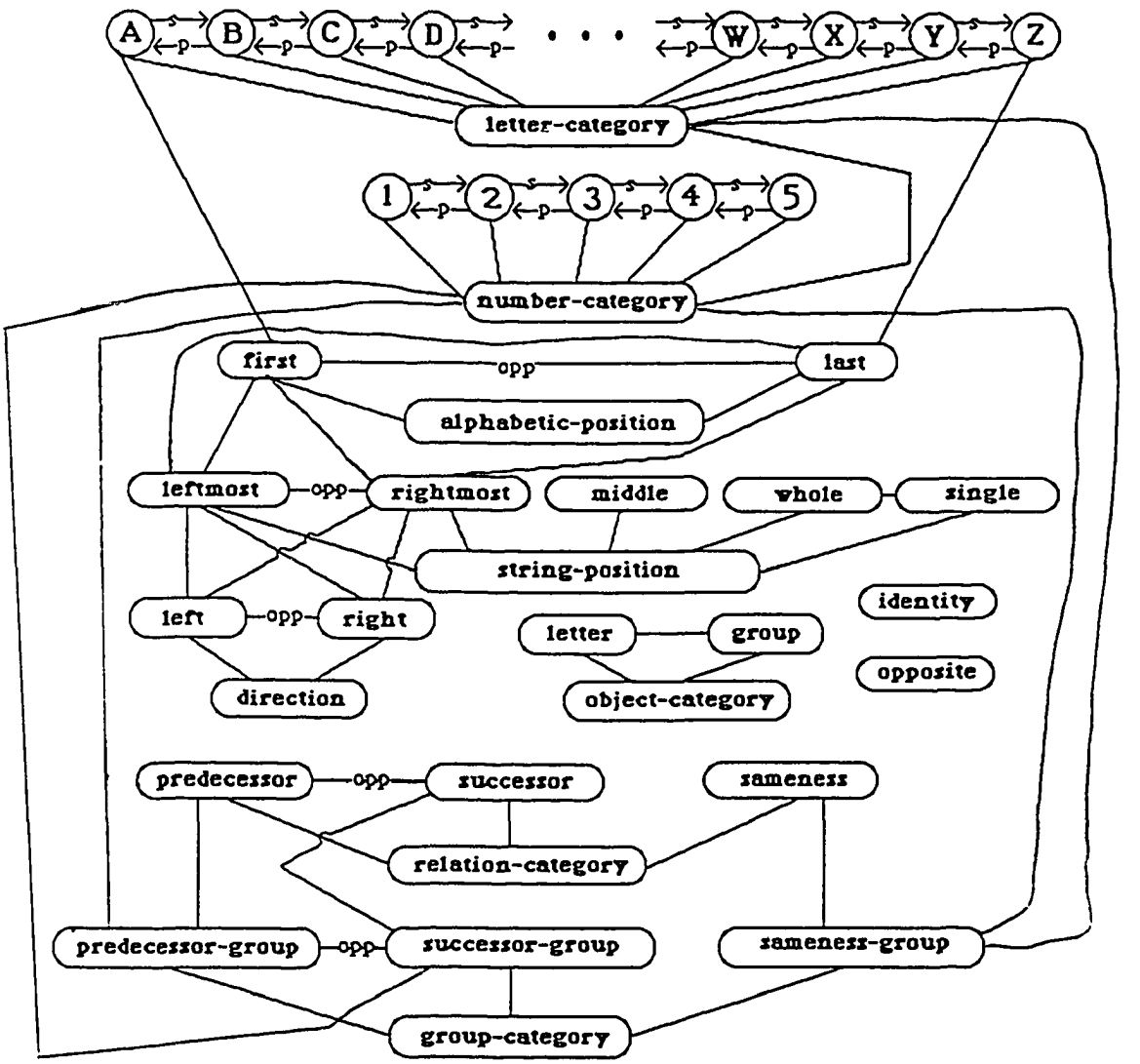


Figure 3.3: Copycat's Slipnet.

- The two possible types of objects in strings: *letter* and *group*.
- The two distinguished positions in the alphabet: *first* and *last*.
- The three possible types of bonds that can be built between objects: *predecessor*, *successor*, and *sameness*.
- The three types of groups that can be made out of related objects in a string: *predecessor-group*, *successor-group*, and *sameness-group*.

In addition, the network includes nodes representing the various *categories* of descriptions:

- *letter-category* (linked to the the letter-category nodes, *A–Z*);
- *number-category* (linked to the number-category nodes, *1–5*);
- *string-position* (linked to the three string-position nodes, *leftmost*, *middle*, and *rightmost*);
- *direction* (linked to the two possible spatial directions, *left* and *right*);
- *object-category* (linked to the two types of objects, *letter* and *group*);
- *alphabetic-position* (linked to the two distinguished alphabetic positions, *first* and *last*);
- *bond-category* (linked to the three possible bond categories, *predecessor*, *successor*, and *sameness*); and
- *group-category* (linked to the three possible group categories, *predecessor-group*, *successor-group*, and *sameness-group*).

Finally, there are the nodes *identity* and *opposite*, which label relationships in the Slipnet (any node has an implicit identity relation to itself) and are used to label certain concept-mappings underlying correspondences. Note that there is a distinction in the system between *sameness* and *identity*: the former is a type of relation between letters or groups—actual objects in the Workspace—and the latter is a type of relation between nodes in the Slipnet. I found it necessary to make this distinction for various purposes in the current version of Copycat, but I am not sure that it is really a proper distinction to make.

The following is a list of the nodes in order of increasing conceptual depth (nodes in a single line have equal conceptual-depth values). The conceptual-depth value for each node is permanent (it does not vary during a run or from run to run) and is assigned by me (the exact values are given in Appendix B). There was no formal method for assigning these rankings; they were decided by a combination of intuition, trial-and-error, and some arbitrariness, and are not necessarily optimally tuned in the current version of the program. An experiment on the program that involved modifying these values will be described in Chapter 7.

- *A* through *Z*;
- *letter*;
- *letter-category*;
- *1* through *5*;
- *leftmost*, *rightmost*, *middle*, *whole*, *single*, *left*, *right*;
- *predecessor*, *successor*, *predecessor-group*, *successor-group*;
- *first*, *last*;
- *number-category*;
- *string-position*, *direction*;
- *alphabetic-position*, *bond-category*, *group-category*;
- *sameness*, *sameness-group*;
- *object-category*, *identity*, *opposite*.

There are four main classes of links in the Slipnet (the different classes are not labeled in the diagram):

- Category links, which relate the *types* of descriptions that can be made to the various possible descriptors of that type (e.g., relating the 26 letters to *letter-category*, or the three types of bonds to *bond-category*).
- Instance links, the inverse of Category links (e.g., relating *letter-category* to the 26 letters).

- Has-Property links, which associate certain nodes with properties the concept has. In the current version of Copycat, there are only two such links: the link from *A* to *first*, and the link from *Z* to *last*.

- Lateral links, which represent various non-hierarchical relationships among nodes. There are two types of lateral links: those whose relationships represent potential slippages, and those whose relationships are not possible slippages. The possible slippage links include:

  - *Opposite* links (labeled "opp"): *leftmost* ↔ *rightmost*, *left* ↔ *right*, *first* ↔ *last*, *predecessor* ↔ *successor*, and *predecessor-group* ↔ *successor-group*.
  - Various unlabeled links encoding various associations: *letter-category* ↔ *number-category*, *letter* ↔ *group*, and *single* ↔ *whole*.

  The lateral links that do not represent possible slippages include:

  - *Successor* and *predecessor* links between letter and number nodes (labeled "s" and "p").
  - Links between the *direction* and *string-position* nodes (e.g., *right* ↔ *rightmost*).
  - The links *first* ↔ *leftmost*, *first* ↔ *rightmost*, *last* ↔ *leftmost* and *last* ↔ *rightmost*. In each link, both nodes refer to extremities, one to a spatial extremity in a string, the other to an extreme position in the alphabet.
  - Links between the various types of bonds and the associated types of groups (e.g., *predecessor* → *predecessor-group*).
  - The links *predecessor-group* → *number-category*, *successor-group* → *number-category*, and *sameness-group* → *number-category*. These links encode the relatively weak associations between the three group-categories and the notion of *length*. Thus, when groups are formed and the corresponding nodes are activated, a small amount of activation is spread to *number-category*, which creates some possibility that the lengths of groups will be perceived. This is illustrated in the screen dumps given in the next chapter.
  - The link *sameness-group* → *letter-category*. In the current version of the program, sameness groups (e.g., the ii group in **iijjkk**) are the only kind of groups that can have letter-category descriptions (e.g., *I*) attached.

As was described in the previous section, the lengths of *labeled* links (namely, the *opposite*, *predecessor*, and *successor* links) decrease when the node corresponding to the label is activated. All other links have lengths that do not change over the course of a run. As was said earlier, the network does not retain any changes from run to run; all node activations and link-lengths are reset to their initial values at the beginning of each new run. The initial lengths of labeled links and the fixed lengths of other links are, as are nodes' conceptual-depth values, assigned by me, based on intuition and trial-and-error, and are not necessarily optimally tuned.

In the current version of Copycat, slippages can be made only between nodes connected by a single lateral slippage link. This is the source of some rigidity in the current program. In principle, the program should be able to perceive relations and make slippages between nodes separated by any number of links, given sufficient pressure. For instance, to solve "**abc** ⇒ **abd**, **ace** ⇒ ?", the program would need to be able to perceive "double successor" relations between the letters in **ace** even though there are no explicit "double successor" links in the Slipnet—and so Copycat cannot currently solve this problem. It should be pointed out that even though all possible slippages are potentially present (as is the slippage *wife* ⇒ *husband* in our minds), it takes pressure to make them, as well as pressure for the concepts involved to become relevant. Many slippages are not plausible *a priori*. As will be seen in the runs given in Chapter 4, the program virtually never makes *opposite* slippages without pressure to do so; in fact, in the absence of pressure, it barely even considers them. Again, it requires sufficient pressure of the right sort in order for certain concepts to become relevant and for certain slippages to be made.

The fact that there are only a small number of possible slippages in the Slipnet also may seem unrealistic, but it should be emphasized that Copycat's small number of relatively simple concepts are meant to *stand for* the large number of more complex concepts in a person's mind. Copycat's concepts are meant to capture in an idealized form some of what is interesting about real-world concepts. Thus it is essential that the program, as much as possible, avoid taking advantage of the facts that it has only a small number of concepts and that each problem has only a small number of elements. The program never searches explicitly through all the nodes and links in its network. Instead, codelets use nodes that become relevant, and make slippages that become plausible, in response to pressures arising both from structures that previously run codelets have built and from existing associations in the network.

Several nodes in the Slipnet (e.g., *successor*, *predecessor*, *successor-group*, and *predecessor-group*) are able, when active, to post specific top-down codelets to the Coderack. The task of such a top-down codelet is to specifically seek instances of its parent node in the Workspace (e.g., to look for successor relations). A list and description of these top-down codelets will be given in a later section.

The combination of the mechanisms discussed here—dynamic, context-dependent activation (representing perceived relevance) and link-lengths (representing perceived conceptual proximity) along with top-down pressure from activated nodes (in the form of codelets seeking instances of those nodes)—results in a model of concepts as "active symbols" (Hofstadter, 1979, Chapter 11; 1985d). Concepts in the Slipnet are active and dynamic, rather than passive and static: they are emergent rather than explicitly defined, they change in response to what is perceived in a given situation and adapt themselves in appropriate ways to different situations. When activated, concepts (e.g., *successor*, or *group-length*) in turn attempt to further instantiate themselves (via top-down codelets) in the current situation; in doing so, they have to compete against each other for the resources of the system (i.e., nodes compete indirectly, via codelets, for running time and for locations to build structures corresponding to instances of themselves).

3.4 Perceptual Structures

3.4.1 What the Program Starts Out With

Although I call the structures described in this section "perceptual structures", the word "perceptual" here is meant to be taken in the same spirit as the phrase "high-level perception"—that is, to refer to the non-modality-specific perceptual processes that occur in the mind when it tries to form an interpretation of a situation, be it a visual scene, a spoken sentence, or an abstract situation such as the Iran–Contra affair. Thus the term "perceptual structure" is meant to be general: it can refer both to modality-specific mental structures such as the structures constructed by Hearsay-II corresponding to phrases, words, syllables, and phonemes, or to abstract mental structures such as the perceived chunk "the Watergate burglars" or a mental correspondence between Reagan and Nixon.

At the beginning of a run, Copycat is given the three strings of letters; its initial knowledge about each component letter consists only of the letter's *letter-category* (e.g., **a** is an instance of category *A*), its *object-category* (i.e., **a** is a *letter*, as opposed to a *group*), its *string-position* (e.g., *leftmost*), and which letters are adjacent to it in its string. Only the

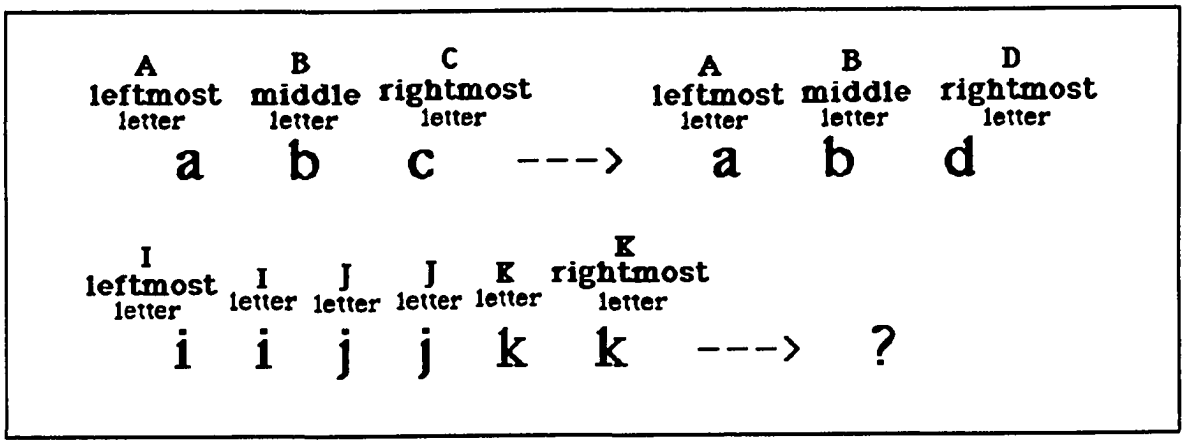


Figure 3.4: The initial descriptions given to the letters in "**abc** ⇒ **abd**, **iijjkk** ⇒ ?".

leftmost, rightmost, and middle letters (if there is one) have *string-position* descriptions (*middle* is used to describe only the single middle object in a string of three objects: e.g., the **b** in **abc**, or the group **jj** in **iijjkk** can both be seen as *middle* objects).

The three strings are presented to the program with no preattached bonds or preformed groups. It is thus left entirely to the program to build up perceptual structures constituting its understanding of the problem in terms of concepts it deems relevant.

Figure 3.4 displays the initial descriptions given to the letters in the problem "**abc** ⇒ **abd**, **iijjkk** ⇒ ?", before the beginning of a run. In the figure, the large boldface lowercase letters are objects in the Workspace, and the descriptions of each letter are listed above it. A description actually consists of two parts: a *descriptor* (e.g., *leftmost*) and an *description-type* (for *leftmost* this would be *string-position*) that names the specific facet (of the object) that is being described.[2] In the figure, only the descriptors are displayed.

---

[2] The structure of a description (e.g., "*string-position*: *leftmost*") is similar to that of a slot and filler in a frame-based representation. However, the words "slot" and "filler" imply that there is a ready-made slot (e.g., *string-position*) attached to the object, waiting to be filled. In Copycat, this is not the case. When the program adds a new description to an object (e.g., *alphabetic-position*: *first*), it is adding both the slot and the filler; the slot did not exist ahead of time. Thus new slots can be added to objects as new concepts (e.g., *alphabetic-position*) become relevant.

For example, the **a** in the initial string has three descriptions: "*letter-category: A*", "*string-position: leftmost*", and "*object-category: letter*". For each description, both the descriptor and the description-type are names of Slipnet nodes.

As can be seen, each letter in the string **abc** happens to have three descriptions, but many letters in **iijjkk** do not have a *string-position* description.

In the figure, a descriptor name in boldface indicates that that description is *relevant*, and thus visible to codelets. A description is relevant only when the Slipnet node corresponding to its description-type is fully active (e.g., *leftmost* is relevant because *string-position* is fully active). Thus the relevance of a description in the Workspace is dynamic and context-dependent, since it changes with the activation of the description-type node, which depends on what has been perceived. In the figure, all the descriptors except *letter* are relevant.

The description-type nodes *letter-category* and *string-position* are initially set to be fully activated, and their activation is clamped (i.e., held constant) for a certain number of time steps. That is, descriptions of these two types are assumed to be relevant, *a priori*. However, this can change over the course of a run: if descriptions of these types do not turn out to be useful, the activation of these description-type nodes will eventually decay, and the corresponding descriptions will no longer be perceived as relevant (this will be seen in some of the runs given in Chapter 4). The program is thus initially biased to assume that certain concepts are relevant, so that *some* aspects of the letters will be visible to early codelets. This reflects the notion that a given situation will have aspects that are *a priori* clearly apparent (e.g., your friend Greg is driving). However, these biases shift in context-dependent ways as a run proceeds, as new structures are built, and as new information is uncovered about what should be considered relevant to the problem at hand.

When a group is formed by the program, it becomes a new object in its own right, and is automatically given the same default types of descriptions that a letter is initially given (e.g., a *string-position* description), if they apply to it. Also, a probabilistic decision is made whether or not to add a *length* description; the longer the group, the less likely it is a description of its length will be explicitly attached to it; in other words, the length of a short group is more easily and immediately perceived than that of a longer group. The probability of adding a length description is generally low, unless group lengths have already been deemed to be relevant, in which case it goes up significantly. Just as for descriptions attached to letters, a group's descriptions lose relevance if they do not turn out to be useful. As will be seen, new descriptions can also be added to an object by

description-making codelets long after the object is created.

Thus, objects are given certain descriptions by default, other descriptions can be added later on if they seem called for, and descriptions lose relevance if they turn out not to be of much use. This is similar to Barsalou's (1989) account of how people construct mental representations: every time a representation (e.g., "frog") is constructed in working memory, certain context-independent, highly accessible descriptions tend to be automatically activated (e.g., frogs are green, frogs move by hopping, etc.), though this information might later be inhibited if it turns out to be irrelevant in the current context (e.g., a French restaurant). Other less-immediate information becomes incorporated only because of its relevance in the current context (e.g., frogs' legs are edible).

Some descriptions are *distinguishing*—that is, they serve to distinguish an object from others in its string (e.g., in the string **abc**, the description *rightmost* distinguishes the **c** since no other object has it, but the description *letter* doesn't distinguish the **c**). The notion of distinguishing descriptions will be employed in the following sections describing how certain codelets use descriptions to build structures.

3.4.2 General Description of Structure-Building

In order to formulate a solution, Copycat must use the concepts it has to make sense of each string as well as to find a set of correspondences between the initial and target strings. To accomplish this, the program gradually builds various kinds of structures in the Workspace that represent its high-level perception of the problem, similar to the way in which Hearsay-II built layers of increasingly abstract perceptual structures on top of raw representations of sounds. (A more detailed comparison between Copycat and Hearsay-II will be given in Chapter 8.) These structures correspond to Slipnet concepts of various degrees of conceptual depth being brought to bear on the problem, and accordingly, each such structure is built of parts copied from the Slipnet.

As was said earlier, the types of structures that Copycat is able to build in its Workspace are the following: *descriptions* of objects (i.e., of letters or groups), *bonds* between objects within a string (the current version of Copycat can build bonds only between spatially adjacent objects), *groups* of objects within a string, *correspondences* between objects in different strings, a *rule* describing the change from the initial string to the modified string, and a *translated rule* describing how the target string should be modified to produce an answer string. Structures in the Workspace can be built and destroyed, although the more built-up,

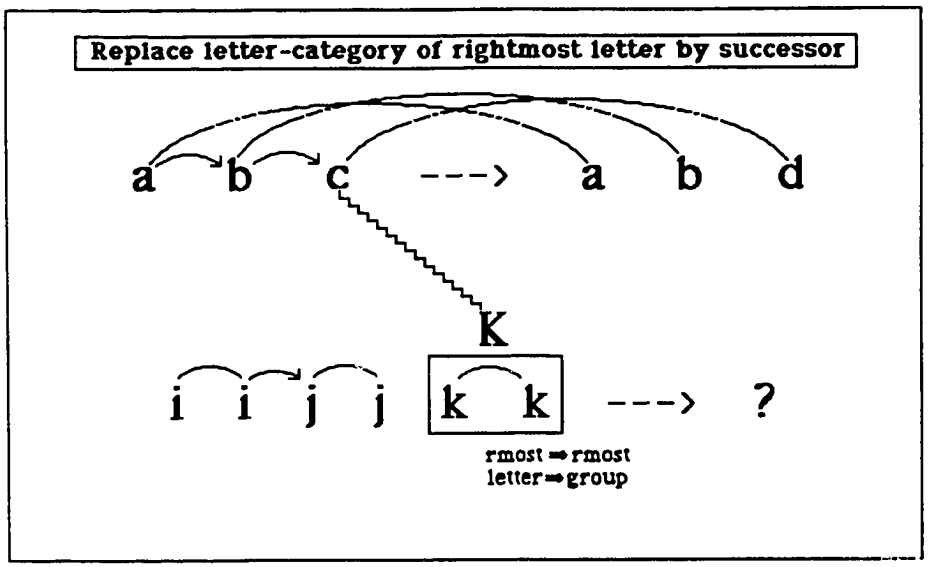


Figure 3.5: A possible state of Copycat's Workspace, with several types of structures shown.

complex, and mutually interrelated the structures become (causing the temperature to fall), the more one hesitates to destroy them.

Figure 3.5 displays a possible state of Copycat's Workspace, where several bonds (arcs between letters), a group (rectangle around the two k's), a correspondence (jagged line from the c to the group of k's), and a rule (shown at top of figure) have been built. (Descriptions attached to letters and groups are not displayed in this figure.) Note that each successor and predecessor bond has a spatial direction (indicated by an arrow on the arc), whereas sameness bonds have no direction. Once built, a group acts as a unitary object much like a letter: it now can itself be an element in a bond, group, and correspondence. (The group of two k's is marked on top by a single K, which gives the letter-category of this group.) The correspondence from the c to the group K is based upon two *concept-mappings* (listed beneath it) between descriptors of the c and descriptors of the group: *rightmost* ⇒ *rightmost* and *letter* ⇒ *group*. These reflect the view that the group K plays the same role in the target string that the c plays in the initial string—namely, both are rightmost in their respective strings. However, in order to make that mapping, a conceptual slippage from *letter* to *group* must be made. (The letter-categories of the two corresponding objects (*C* and *K*) are ignored in this correspondence, since in the Slipnet there is no close relation between these nodes.)

In order to formulate a rule, the current version of Copycat assumes that the initial and modified strings (e.g., **abc** and **abd**) will be the same length, and that exactly one letter will be changed. It cannot presently deal with more complex changes from the initial to the modified string; for the purposes of this project, the concentration has been more on dealing with interesting mappings between the initial and target strings.

This one-letter-change restriction of course limits the space of problems Copycat can currently deal with, but the space of interesting problems of this type is still so large that the current program barely scratches the surface. Codelets do almost no examination of the modified string; the program instead concentrates on the tasks of making sense of the initial and target strings, and constructing a mapping between them. The only analysis done of the modified string is to spell out the one-to-one letter correspondences between it and the initial string (shown as horizontal arcs in Figure 3.5), and to determine what, if any, relationship there is between the changed letter and its replacement. If there is any such relationship (as there is in **abc** ⇒ **abd**, namely successorship), then a description reflecting that relationship is added to the replacement letter's (here, d's) list of descriptions.

The rule is formed by a codelet that fills in the template "Replace _____ by _____" with descriptors of the changed letter and its replacement. These descriptors are chosen probabilistically, with a bias towards choosing descriptors of greater conceptual depth. Thus, although both "Replace letter-category of rightmost letter by successor" and "Replace letter-category of rightmost letter by *D*", or even "Replace *C* by *D*", are all possible, the first is more likely to be formed than the latter two, since descriptors *rightmost* and *successor* are more general than *C* and *D*. (There are some analogy problems in which one of the latter two rules would be preferable to the first, and accordingly, the rule-preference function is actually not as straightforward as described above. This will be illustrated in some of the variant problems discussed in Chapter 5.)

3.4.3 How Copycat Decides to Stop

Copycat decides probabilistically when to translate the rule and come up with an answer. This works as follows. The formation of a rule triggers the program to begin posting "rule translator" codelets to the Coderack. The job of a rule-translator codelet is to translate the rule—according to whatever slippages have been made—in order to apply it to the target string and produce an answer. When a rule translator is chosen and runs, the first thing it does is decide probabilistically whether or not it really should go ahead and translate the

rule, or whether it should "fizzle" instead. One might think at first that the decision should be based entirely on the temperature, reasoning that if the temperature is low, then the program has a good sense of what is going on in the problem and it should go ahead and try to produce the answer, and if the temperature is high, then not enough structure has been built yet and the program should keep trying to build more structures so as to improve its understanding of what is going on. However, suppose this strategy were adopted, and then the program were given a problem like "**abc ⇒ abd, wqlh ⇒ ?**", where there are no structures to be built in the target string. The temperature will never get very low on this problem, and if the decision to stop were based only on temperature, the program might keep attempting to build structures for a long time before finally making the low-probability decision to quit. Instead, the program should be able to sense in some way that it has attempted for long enough to make sense of the problem and that it is unlikely to find any more structures; at this point the program should be more likely to give up at a high temperature than it would if the outlook were more promising.

In Copycat this works as follows. A rule translator's decision whether or not to translate the rule (thus stopping the program) depends both on the temperature and the amount of structure that has already been built (recall that the temperature is a function of not just the amount of structure that has been built, but of its *quality* as well, so it is possible for there to be a fair amount of structure and for temperature to still be high, if the structure is weak). There are three possibilities:

1. The temperature is low. This means that a reasonable amount of high-quality structure has been built, and that the program should go ahead and try to produce an answer at this point. In this case, the rule translator has a higher probability of deciding to go ahead and translate the rule.

2. The temperature is high, and not much structure has been built (this would be true for "**abc ⇒ abd, wqlh ⇒ ?**"). In this case, the rule translator again has a higher probability of deciding to go ahead and translate the rule. The reason is that rule translators tend not to run until many other codelets have had a chance to run and build structure (this is not deterministic, but it is statistically true, since the posting of rule translators is triggered only when a rule has been formed). Thus if a rule translator is running and finds that very little structure has been built, then the assumption is that the program would have already had a chance to build structures

if there were any to build, so there must not be much structure in the problem. The program should thus give up and go ahead with producing an answer, since it is unlikely to find a better one.

3. The temperature is high, but a fair amount of structure does exist (this might be the case in "**abc ⇒ abd, ijklmnopqrs ⇒ ?**", before the program builds all the bonds in the target string). Here Copycat assumes that there *is* structure to be found, but the program just hasn't yet found all of it or perhaps the structure that it *has* found could be changed for the better. In this case, the rule translator is likely to decide to fizzle, allowing the program to continue exploring ways of building structures.

Even in cases 1 and 2, the probability of going ahead and translating the rule is fairly low, though it is higher than in case 3, and in general many rule-translator codelets have to run before one succeeds in translating the rule. Thus, even in the first two cases, it takes pressure—in the form of many rule translators—in order for the program to stop. Stopping is more likely in the first two cases than it is in the third, but the desired behavior of the program in deciding when to stop emerges from the statistics of many codelets rather than the individual action of a single codelet.

Once the rule has been translated, the program stops running codelets, and creates its answer to the problem by taking the rule describing the initial-string-to-modified-string change and *translating* it according to any slippages underlying the correspondences between the initial and target strings. In the example displayed in Figure 3.5, the rule would be translated as "Replace letter-category of rightmost *group* by successor" (using the slippage *letter* ⇒ *group*), yielding answer **iijjll**.

3.4.4 Strengths of Structures

A structure's *strength* at a given time is used by codelets to make probabilistic decisions, such as whether or not to continue evaluating that particular structure, what urgencies should be assigned to codelets that will further evaluate it, and whether that structure should win a fight against an existing incompatible structure. Here I describe in general terms how the strength of each type of structure is calculated.

The strength of a structure is a function of both internal and external aspects; that is, of both intrinsic aspects of the structure and aspects of it in terms of its relation to other structures that have been built. The various aspects that contribute to a structure's

strength reflect various pressures that interact in the construction and evaluation of that type of structure.

- **Descriptions:** The strength of a description is a function of the following.
  1. *The conceptual depth of the descriptor* (e.g., *rightmost*), reflecting a bias towards more descriptions of greater conceptual depth.
  2. *The activation of the description-type* (e.g., *string-position*), reflecting a bias towards building relevant types of descriptions.
  3. *The "local support" of the description*, i.e., the number of other descriptions of the same type (e.g., *string-position* or *length*) in the same string, reflecting a bias towards building types of descriptions that have already been used in the problem. For example, if a length description has already been attached to the **rr** group in **mrrjjj**, then a proposed length description of the **jjj** group would have some local support. In other words, as far as descriptions are concerned, there is safety in numbers.
- **Bonds:** The strength of a bond is a function of the following.
  1. *The current strength of its bond-category.* Each type of bond has a certain intrinsic strength (e.g., *sameness* is stronger than *successor* or *predecessor*), though this can be changed by activation (e.g., when *successor* is active, successor bonds become stronger).
  2. *The bond's local support*, i.e., the number of other bonds of both the same category (e.g., *successor*) and spatial direction (e.g., *right*) that currently exist in the same string. This formula again reflects the philosophy of "safety in numbers".
- **Groups:** A group is always associated with a bond-category (e.g., the **kk** sameness-group is associated with the bond-category *sameness*). The strength of a group is a function of the following.
  1. *The current strength of the group's bond-category* (e.g., sameness groups are generally stronger than successor and predecessor groups).
  2. *The group's length* (the longer, the stronger).
  3. *The group's local support*, i.e., the number of other groups in the same string with the same group-category and direction. For example, once the group **kk**

has been built in the string **iijjkk**, a potential **jj** group becomes stronger because of it, since both are sameness groups (which have no spatial direction).

- **Correspondences**: A correspondence between two objects is based on a set of concept-mappings between descriptors of the two objects, as in the example displayed in Figure 3.5. Concept-mappings are either *identities* (e.g., *rightmost* ⇒ *rightmost*) or *slippages* (e.g., *letter* ⇒ *group*). When a correspondence is made, some descriptors can be ignored (in that example, the descriptors *C* and *K* are ignored since these nodes are not close enough in the Slipnet, and thus play no role in supporting the correspondence). These classifications of concept-mappings are similar to Holyoak's (1984) taxonomy of mapping relations: identity concept-mappings correspond to his "mapped identities", slippages correspond roughly to his "structure-preserving differences" (though his taxonomy doesn't involve slippages in the same way they are used in Copycat, and the notion of structure-preserving differences is by no means as central to his taxonomy as slippage is to Copycat), and ignored descriptors correspond to his "structure-violating differences". In Copycat, the strength of a correspondence is a function of the following.

  1. *The number of concept-mappings it is based on*, reflecting the idea that the more similarities there are, the stronger the correspondence.
  2. *The proximity of the two nodes in each concept-mapping*, reflecting the idea that the stronger the similarities, the stronger the correspondence.
  3. *The conceptual depth of the two nodes in each concept-mapping*, reflecting the idea that the deeper the similarities, the stronger the correspondence. Even though the inclusion of deep similarities adds strength to a correspondence, there is also a pressure resisting *slippages* between descriptors with a high degree of conceptual depth (such as *first* and *last*), since better analogies are generally ones in which shallow aspects slip while deep aspects remain invariant. This conflict of pressures will be discussed and demonstrated in several of the runs in the next chapter—in particular, in the run of the problem "**abc** ⇒ **abd**, **xyz** ⇒ ?".
  4. *The internal coherence of the correspondence*—that is, the degree to which the underlying concept-mappings support each other. Two concept-mappings support each other (or, in other words, are *conceptually parallel*) if their corresponding descriptors are conceptually related (i.e., the nodes are linked in the Slipnet)

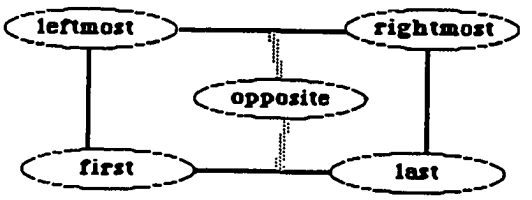


Figure 3.6: Illustration of internal coherence of concept-mappings *first* ⇒ *last* and *leftmost* ⇒ *rightmost*.

and if the two concept-mappings represent the same *relationship* (e.g., *opposite*). Thus *leftmost* ⇒ *rightmost* and *first* ⇒ *last* support each other, since the pairs *leftmost* and *first*, and *rightmost* and *last*, are conceptually related (the nodes in each pair are linked in the Slipnet), and both concept-mappings are *opposite* slippages, so a correspondence including both is internally coherent. This is illustrated in Figure 3.6 (As will be seen in the run of Copycat on "**abc** ⇒ **abd**, **xyz** ⇒ ?", this internal coherence is one of the reasons the **a**–**z** correspondence can come to be seen as strong.)

5. *The size of the objects involved in the correspondence.* There is a bias towards correspondences that connect larger parts of the two strings (i.e., correspondences involving groups tend to be stronger than correspondences involving letters, and correspondences involving large groups–in particular, whole-string groups—tend to be stronger than correspondences involving small groups). This reflects a desire for mappings involving large, coherent parts of the two strings, which is similar to Gentner's (1983) notions of "structure-mapping" and "systematicity" (these will be discussed in detail in Chapter 8).

6. *The strengths of the other correspondences that support the given correspondence.* Two correspondences support each other if a concept-mapping in one supports (i.e., is conceptually parallel to) a concept-mapping in the other. For example, two correspondences containing (respectively) the concept-mappings *rightmost* ⇒ *rightmost* and *leftmost* ⇒ *leftmost* support each other. In contrast, the concept-mappings *rightmost* ⇒ *leftmost* and $C \Rightarrow C$ do not support each other but are not incompatible with each other, while *rightmost* ⇒ *leftmost* and *right* ⇒ *right* conceptually contradict each other and are thus incompatible (in Copycat,

incompatible correspondences cannot exist simultaneously). A requisite for any strong analogy is a set of strong, mutually supporting correspondences.

These notions of internal coherence and mutual support between correspondences are related to some of the ideas put forth by Thagard (1989) about "conceptual coherence" in scientific theorizing. Many components of the definition of correspondence strength have counterparts in Gentner's (1983) structure-mapping theory and in Holyoak and Thagard's (1989) ACME program. These comparisons will be discussed further in Chapter 8.

- **Rules:** The strength of a rule is a function of the following.

  1. The conceptual depth of the descriptors used in the rule. For example, as was mentioned earlier, the rule "Replace letter-category of rightmost letter by successor" is stronger than "Replace *C* by *D*".

  2. *How the changed letter in the initial string has been mapped to the target string.* For example, suppose that, in the problem "**abc** ⇒ **abd**, **cccc** ⇒ ?", the **c** in **abc** is seen as corresponding to the entire group of four **c**'s in the target string. The rule "Replace *C* by *D*" is more compatible with this "worldview" than is the normally stronger "Replace letter-category of rightmost letter by successor", since the descriptor *rightmost* plays no role in the **c**–**cccc** mapping. The strength formula takes such mappings into account in determining how compatible the rule in question is with the structures that have been built so far.

It should be pointed out that the strength calculations for structures often involve looking at *all* the other structures of that type in the same string. Such a complete search might be implausible in a complex, real-world situation, even though the number of mental structures a person imposes on a given situation is small compared with the number of objects in the situation.[3] The role of these functions is therefore not to propose detailed psychological mechanisms for how these strength values are computed, but rather to produce plausible numbers that can be used in the mechanisms that we *are* proposing, as well as to spell out the pressures that are involved in coming up with these numbers.

---

[3] Another problem is that it is in general hard to define real situations in terms of discrete objects; what is or is not a single object is blurrier in the real world than in the more cut-and-dried letter-string microworld.

3.4.5 Importance, Happiness, and Salience of Objects

Each letter or group in the initial and target strings has three time-varying values associated with it: importance, happiness, and salience. As will be seen, these values are used in various ways by the program as it runs.

- The *importance* of an object is a function of (1) how many relevant descriptions it has (recall that a description (e.g., *string-position: leftmost*, is relevant if its description-type, e.g., *string-position*, is fully active) and (2) how active the corresponding descriptors (e.g., *leftmost*) are. For example, the leftmost **i** in **iijjkk** would usually have a higher importance than its right neighbor, since the former has a *string-position* description and the latter does not (in general, objects on the edges of strings have higher importance than internal objects, though if *string-position* descriptions happened to become irrelevant, this difference would vanish). The intuition here is that the objects perceived to be important in a situation are the ones that are easiest to describe (i.e., have many relevant descriptions) and whose descriptors are most visible (i.e., highly activated). In addition, once the changed object in the initial string (e.g., the **c** in **abc** ⇒ **abd**) has been identified, its importance is raised, since it plays the leading role in defining the relationship between the initial string and the modified string. Also, the importance of any object inside a group (e.g., the individual **k**'s in the group **K**) is lowered, since when objects are grouped, they follow a utilitarian philosophy, partially relinquishing their individual interests for the good of the larger unit.

- The *happiness* of an object depends on how well it fits into the overall structuring of its string and the mapping from the initial string to the target string, as well as how good that structuring and mapping seem. Thus, an object's happiness is a function of the strengths of the structures (bonds, groups, and correspondences) attached to it. For example, in Figure 3.5, the **c** in **abc** is happier than the **b** because it has a correspondence to something in the target string. Each object starts out with certain slots available for bonds, groups, and correspondences, and its happiness at a given time depends on how well those slots are filled.

- The *salience of* an object—in effect, the object's attractiveness to codelets—is a function of its importance and its unhappiness (the inverse of its happiness). Increased

importance leads to higher salience because important objects merit attention in their own right. Increased unhappiness leads to higher salience because unhappy objects need attention from codelets in order to increase their happiness.

If an important object is very happy, it doesn't need much attention (e.g, the **c** in Figure 3.5 doesn't need much attention). Likewise, if an unhappy object is not very important, it doesn't merit the same attention that should be given to an equally unhappy important object. For example, the rightmost **j** in Figure 3.5 is not as important as the leftmost **i**, so the former would tend to get less attention, even though both are equally unhappy. (There are a number of real-world counterparts of these relations among one's importance, one's unhappiness, and the amount of attention one gets; for instance, consider the relative amount of coverage in the media given to crimes against various members of society.)

Finally, the *temperature* at any time is a weighted average of the unhappinesses of all objects, where each object's unhappiness is weighted by its importance. Thus, important objects have more of an effect on temperature than unimportant ones.

The details of how temperature is calculated are given in Appendix B.

3.5 Codelets

3.5.1 General Comments about Codelets and Structure-Building

Earlier in this chapter I gave an overview of how codelets cooperate and compete with each other to gradually build up structures. In this section I will describe what the different types of codelets are, and discuss in more detail how structure-building takes place.

It is worth making clear the distinction between codelet *types* and codelet *instances*. The program has a fixed number of codelet types, which are pre-written pieces of code, but it is *instances* of these platonic types that are placed on the Coderack (sometimes with specific arguments filled in) and that run. As will be seen, there are 24 different types of codelets, but, at a given time, the Coderack can contain a much larger number of instances waiting to run (and rarely are instances of all 24 types present at the same time), with various types being represented by various *densities* of instances, the densities being a result of what has happened in the run so far.

In general, each codelet type (with a few exceptions; see below) is associated with some aspect of evaluating or building a particular type of structure (a description, bond, group,

correspondence, or rule). As was described earlier, structure-building in Copycat is broken up into small steps, so that a parallel terraced scan of possibilities can be carried out. In the current version of the program, the evaluation and building of any individual structure is carried out by a chain of three codelets. First, a *scout* codelet probabilistically chooses one or more objects, where the choice is based on the relative saliences of the various objects in the problem. Thus, objects of high salience tend to be chosen more often (and thus paid more attention to) than objects of lower salience. The scout then determines whether or not it is possible to attach its particular type of structure to the chosen objects. As was mentioned before, there are two types of scout codelets: a *bottom-up scout* is willing to consider any variety of the particular structure-type it is looking for (e.g., the bottom-up-bond-scout codelet will consider bonds of any type), whereas a *top-down scout*, which is posted by some active Slipnet node (e.g., *successor*), sees if it can attach the specific structure associated with that node (e.g., a successor bond) to the chosen objects. In summary, a scout codelet "tests the waters" for a possible structure. If the scout codelet discovers any reason for building its structure, it places a *strength-tester* codelet on the Coderack, giving it the proposed structure as an argument, and assigning it an urgency based on certain somewhat superficial and quickly evaluated aspects of the structure (if no reason is found to attach the structure to the chosen objects, then no strength-tester codelet is posted, and the chain fizzles at this point). When the strength-tester runs, it calculates the strength of the given structure, and, based on this calculation, decides probabilistically whether or not to post a *builder* codelet. If the decision is "yes", a builder codelet is placed on the Coderack, its urgency being a function of the structure's strength; if "no", the chain fizzles at this point. When the builder codelet runs, it tries to build the structure, fighting against incompatible already-existing structures if necessary. The outcomes of the fights are decided probabilistically on the basis of the competing structures' strengths, and the new structure has to defeat *all* the existing incompatible structures before it can knock any of them down. Thus, when there is more than one strong rival, the odds are against a new structure. However, if the proposed structure wins all the fights, all the rival structures are destroyed, and the new structure is built.

In summary, the three-codelet chain for building a given type of structure goes as follows:

- A *scout* codelet asks, "Is there any reason for building this type of structure in this location?";

- If yes, a *strength-tester* codelet asks, "Is the proposed structure strong enough?";
- If yes, a *builder* codelet tries to build the structure, fighting against competitors if necessary.

This small pathway should not be identified with what was referred to earlier as a "path of exploration". A path of exploration does not involve just one structure, but an entire set of steps leading to an answer, in which a large number of codelets and structures participate. Paths of exploration are defined as any of the possible ways in which the program could structure its perceptions of the problem in order to construct an analogy. Thus possible paths are not laid out in advance for the program to search, but rather are constructed by the program as its processing proceeds, just as in a game of chess, where paths through the tree of possible moves are constructed as the game is played. The evaluation of a given move in a game of chess blurs together the evaluations of many possible look-ahead paths that include that move. Similarly, any given action in building a structure by a codelet in Copycat is a step included in a large number of possible paths toward a solution, and an evaluation obtained by a codelet of a proposed structure blurs together the estimated promise of all these paths.

3.5.2 Codelet Types

Copycat has codelet types to scout out, evaluate, and build all types of structures—descriptions, bonds, groups, correspondences, and rules—as well as to translate rules and to break structures that have been built. The 24 codelet types in Copycat are described below, with the arguments taken by each codelet indicated.

The description here is at a medium level of detail, leaving out some details for the sake of clarity. More detailed descriptions of the various codelet types are given in Appendix C.

Often, a codelet chooses one or more objects to use in attempting to build a structure. The choice of what object or objects to use is probabilistic and is in most cases based on the relative salience of objects in the problem (where more salient objects are more likely to be chosen). Unless stated otherwise, this is what "chooses an object" means in the descriptions given below.

Description-Building Codelets

- **Bottom-up description-scout** (no arguments): A codelet of this type chooses an object—say, the **a** in **abc**—and a description of that object—say "*letter-category*:

A"—and sees if any new description can be attached to the object based on *has-property* links in the Slipnet. For example, this codelet would look for any conceptually close properties of *A*. If *first* were seen as close enough to *A* (a probabilistic decision), the codelet would propose a new description: "*alphabetic-position*: *first*", and would post a description-strength-tester codelet to further evaluate this proposed description.

- **Top-down description-scout** (argument: a description-type node): Codelets of this type represent pressure to build a specific *type* of description. They are placed on the Coderack by an active description-type node (such as *alphabetic-position*), which then becomes the argument. A codelet of this type chooses an object—say, the **a** in **abc**—and sees if a new description of the given type—say, *alphabetic-position*—can be attached to the object. If the object is the **a**, then the codelet can propose the description "*alphabetic-position*: *first*"; if the object were the **b**, then since no alphabetic-position description would be possible, the codelet would fizzle. If such a description can be made, this codelet proposes it and posts a description-strength-tester codelet to continue the evaluation.

  The nodes that can post top-down descriptor-scout codelets are *string-position*, *alphabetic-position*, and *number-category* (which tries to describe groups in terms of their lengths).

- **Description-strength-tester** (argument: a proposed description): This codelet calculates the proposed description's strength, and based on the result, probabilistically decides whether or not to post a description-builder codelet. If so, the urgency of the description-builder codelet is a function of the strength.

- **Description-builder** (argument: a proposed description): This codelet builds the proposed description (if it hasn't already been built by a previous codelet chain).

Bond-Building Codelets

- **Bottom-up bond-scout** (no arguments): A codelet of this type chooses a pair of adjacent objects and sees if there is any bond that can be made between them (e.g., successorship). If so, this codelet proposes the bond and posts a bond-strength-tester codelet to evaluate it.

- **Top-down bond-scout [category]** (argument: a bond-category node):
  Codelets of this type represent pressure to build bonds of a specific *category*. They

are placed on the Coderack by an active bond-category node (*predecessor*, *successor*, or *sameness*) which then becomes the argument. A codelet of this type chooses a pair of adjacent objects and sees if a bond of the given category can be made between them. If so, this codelet proposes the bond and posts a bond-strength-tester codelet to evaluate it.

- **Top-down bond-scout [direction]** (argument: a direction node): Codelets of this type represent pressure to build bonds with a specific *spatial direction*. They are placed on the Coderack by an active direction-category node (*left* or *right*) which then becomes the argument. A codelet of this type chooses a pair of adjacent objects and sees if a bond (of any category) can be made between them in the given direction. If so, this codelet proposes the bond and posts a bond-strength-tester codelet to evaluate it.

- **Bond-strength-tester** (argument: a proposed bond): This codelet calculates the proposed bond's strength, and based on it, probabilistically decides whether or not to post a bond-builder codelet. If so, the urgency of the bond-builder is a function of the strength.

- **Bond-builder** (argument: a proposed bond): This codelet tries to build the proposed bond (if it hasn't already been built by a previous codelet chain), fighting with competitors (e.g., an already existing bond between the two objects) if necessary.

Group-Building Codelets

A group is based on a set of bonds between adjacent objects, all of the same bond-category and direction. The building of groups is triggered only when bonds have already been built. Thus there is no bottom-up group-scout that is willing to look for any kind of group whatsoever; instead, when a bond (e.g., successor) is built, the corresponding bond-category node (e.g., *successor*) is activated, and spreads activation to the node representing the associated group-category (e.g., *successor-group*), which posts top-down group-scout codelets to seek instances of groups of that category.

- **Top-down group-scout [category]** (argument: a group-category node): Codelets of this type represent pressure to build groups of a specific *category*. They are placed on the Coderack by an active group-category node (such as *successor-group*), which then becomes the argument. A codelet of this type chooses a number of adjacent

bonds and sees if they are all of the given category (e.g., *successor*) and in the same direction. If so, the codelet proposes a group based on these bonds and posts a group-strength-tester codelet to evaluate it.[4]

- **Top-down group-scout [direction]** (argument: a direction node): Codelets of this type represent pressure to build groups with a a specific *spatial direction*. They are placed on the Coderack by an active direction-category node (*left* or *right*) which then becomes the argument. This codelet works in basically the same way as the top-down group-scout–category codelet, except that it looks for a group based on adjacent bonds all having the given *direction*, not caring which bond-category they have, so long as they all have the same one.

- **Group-string-scout** (no arguments): Codelets of this type represent pressure to construct a group out of the entire string (not caring which category or direction the group has). The construction of groupings of both initial and target strings as wholes is so desirable for the program that an entire codelet type is dedicated to attempting this task. The codelet sees if there are bonds of the same category and direction that span the string. If so, it proposes a group based on these bonds, and posts a group-strength-tester codelet to evaluate it.

- **Group-strength-tester** (argument: a proposed group): This codelet calculates the proposed group's strength, and based on it, probabilistically decides whether or not to post a group-builder codelet. If so, the urgency of the group-builder codelet is a function of the strength.

- **Group-builder** (argument: a proposed group): This codelet tries to build the proposed group (if it hasn't already been built by a previous codelet chain), fighting with competitors (e.g., already-existing groups containing some of the same objects) if necessary.

Correspondence-Building Codelets

---

[4] Given enough local support for this group-category, this codelet can even propose a group consisting of just a single letter, though in most circumstances, this is unlikely. How single-letter groups get proposed and built will be described in Chapter 4, when a sample run of the program's solution "**abc ⇒ abd, mrrjjj ⇒ mrrjjjj**" is displayed.

- **Bottom-up correspondence-scout** (no arguments): This codelet chooses two objects, one from the initial string and one from the target string. It sees if there are any possible concept-mappings that can be made between descriptors of the two objects. A concept-mapping can be made between two descriptors if they are both relevant (i.e., the description type of each is fully active), of the same description-type (e.g., *string-position*), and are sufficiently close in the Slipnet. For example, the correspondence between the **c** and the group **K**, pictured in Figure 3.5, has two concept-mappings: *rightmost* ⇒ *rightmost* and *letter* ⇒ *group*. The two *letter-category* descriptors, *C* and *K*, were not sufficiently close to each other in the Slipnet for a concept-mapping to be made between them.

  If there is at least one such concept-mapping between *distinguishing* descriptors (e.g., the *rightmost* ⇒ *rightmost* mapping shown in Figure 3.5), then the codelet proposes a correspondence between the two objects including *all* the qualifying concept-mappings (non-distinguishing ones such as *letter* ⇒ *group* come along for the ride), and posts a correspondence-strength-tester codelet to evaluate the proposed correspondence.

- **Important-object correspondence-scout** (no arguments): The task of codelets of this type is to find the target-string counterparts of *important* objects in the initial string. The idea here is to model the way people, when making an analogy, focus on important objects and roles in one situation (e.g., you focus on Ronald Reagan as "the President" in the Iran–Contra situation), and actively try to retrieve the object filling the corresponding role in the other situation (e.g., you actively try to figure out who is "the President" in the Watergate situation).

  To accomplish its task, a codelet of this type chooses an object from the initial string probabilistically, using importance rather than salience as its bias. It then chooses one of the object's descriptions and sees if there is any object in the target string that has the "same" description, taking into account any slippages that have already been made. For example, in the problem "**abc** ⇒ **abd**, **kji** ⇒ ?", this codelet might choose the **a** in **abc**, choose its description *leftmost*, and try to make a correspondence with the leftmost object in **kji**. But if a correspondence has already been made between the **c** and the **k** with the slippage *rightmost* ⇒ *leftmost*, then this codelet will take that into account and consider a correspondence between the **a** and the *rightmost* object in the target string. If the desired target-string counterpart is found, then this

codelet proposes a correspondence between the two objects in the same manner as in the bottom-up correspondence-scout codelet.

- **Correspondence-strength-tester** (argument: a proposed correspondence): This codelet calculates the proposed correspondence's strength, and based on it, probabilistically decides whether or not to post a correspondence-builder codelet. If so, the urgency of the correspondence-builder is a function of the strength.

- **Correspondence-builder** (argument: a proposed correspondence): This codelet tries to build the proposed correspondence (if it hasn't already been built by a previous codelet chain), fighting with competitors (e.g., incompatible correspondences) if necessary.

Rule-Building Codelets

- **Rule-scout** (no arguments): This codelet fills in the rule template (as was mentioned before, the current version of Copycat has only one: "Replace _____ by _____"). To do this, it probabilistically chooses descriptors of the changed letter in the initial string and of the letter in the modified string that replaces it, with a bias towards descriptors with greater conceptual depth. This codelet proposes a rule and posts a rule-strength-tester codelet to evaluate it.

- **Rule-strength-tester** (argument: a proposed rule): This codelet calculates the proposed rule's strength, and based on it, probabilistically decides whether or not to post a rule-builder codelet. If so, the urgency of the rule-builder is a function of the strength.

- **Rule-builder** (argument: a proposed rule) : This codelet tries to build the proposed rule (if it hasn't already been built by a previous codelet chain), fighting with the existing rule, if there is one and if it is different from the proposed rule.

- **Rule-translator** (no arguments): As described in Section 3.4.3, this codelet first decides probabilistically, based on temperature and on how much structure has been built already, whether or not to fizzle without doing anything. If it decides to proceed, it translates the rule according to the translation instructions given in the slippages in the Workspace. Once the rule has been translated, the program proceeds to construct an answer according to the directions in the translated rule, and then halts.

Other Codelets

- **Replacement-finder** (*no arguments*): This codelet chooses a letter at random in the initial string and builds a "replacement" structure between the chosen letter and its counterpart in the modified string. If the replacement involves a change of letter-category (as for the **c** to **d** replacement in **abc** ⇒ **abd**), this codelet marks the initial-string letter as "changed" and gives the corresponding modified-string letter a description describing the change relation (e.g., successorship), if there is one (e.g., if the **c** changed to a **q**, there is no change relation, so no description would be given). Note that the process of finding replacements in the modified string for initial-string letters does not follow the usual three-codelet building process. This is because, as was described earlier, the program assumes a one-to-one letter-to-letter mapping between the initial and modified strings, and thus the initial-string-to-modified-string mapping is trivial to determine. Of course, this assumption severely limits the range of problems that the program, as it now stands, can solve, and this stage will have to be much more complex if the program is to be extended to solve problems with more complex initial-string-to-modified-string changes.

- **Breaker** (no arguments): This codelet's task is to try to break some structure, but the first thing it does is decide probabilistically, based on the current temperature, whether or not it should instantly fizzle (the lower the temperature, the more likely it is to fizzle). If not, it chooses a structure at random and decides probabilistically, as an inverse function of the structure's strength, whether to break the structure.

As was mentioned earlier, codelets, for the most part, are biased to choose salient objects to work on. Recall that the salience of an object is a function of both its importance and its unhappiness. This is related to the "romance" metaphor discussed in the section on Jumbo. Before any bonds, groups, or correspondences are formed, all objects are equally unhappy, so the relative salience of the various objects is determined wholly by their relative importance. But as structures are built, the objects that are "hitched up" become happier, depending on the strength of their ties to other objects, so their salience goes down, which means they are chosen less often by codelets. In terms of the metaphor, the happier the romance (i.e., the happier the objects in a given structure), the less the "flirting" done by the romantic partners (the less the codelets trying to build other structures look at the already "involved" objects, since higher happiness causes lower salience). However, the

more desirable a person (the more important an object), the more flirting is done (the more codelets look at that object, since higher importance leads to higher salience).

3.6 Temperature

Copycat's temperature variable measures the current disorganization in the system's understanding of the problem: the value of the temperature at a given time is a function of the unhappiness of the objects in the problem, which is in turn a function of the amount and quality of perceptual or organizing structure that has been built so far. Thus temperature starts high, falls as structures are built, and rises again if structures are destroyed, if their strengths decrease, or if new objects (i.e., groups) are formed and need to be incorporated into a coherent structuring of the problem. In turn, the value of temperature controls the degree of randomness used in probabilistic decision-making in the system. There are two related ideas here. The first is that when there is little perceptual organization (and thus high temperature), the information on which decisions are based (such as the urgency of a codelet or the strength of a particular structure) is not very reliable, and decisions should be more random than would seem to be indicated by this information. When a large amount of structure deemed to be good has been built (and thus temperature is low), the information is considered to be more reliable, and decisions based on this information should be more deterministic.

The second idea is that early on, when not much is known about the situation to be understood, the system should pursue a large number of parallel explorations, so that enough information can be obtained in order to make *intelligent* decisions later on about what possibilities to focus on. Thus, early on, exploration should be parallel and fairly random (i.e., stochastic with a fairly even distribution), and it should gradually become more and more focused, serial, and deterministic as more becomes understood about the situation at hand. Temperature, by implementing feedback between the quality of the program's understanding and the degree of randomness at a given time, provides a mechanism for achieving this continuous transition. This mechanism will be illustrated in detail in the sample runs of Copycat given in the next chapter.

The solution to the well-known "two-armed bandit" problem (Given a slot machine with two arms, each with an unknown payoff rate, what strategy of dividing one's play between the two arms is optimal for profit-making?) is an elegant mathematical verification of these ideas (an excellent discussion of this solution and its implications is given by Holland,

1975). The solution states that the optimal strategy is at all times to be *willing* to sample either arm, but with probabilities that diverge increasingly fast as time progresses. In particular, as more and more information is gained through sampling, the optimal strategy is to exponentially increase the probability of sampling the better-seeming arm relative to the probability of sampling the worse-seeming arm (note that one never knows with absolute certainty which *is* the better arm, since all information gained is merely statistical evidence). Copycat's parallel terraced scan can be likened to such a strategy extrapolated to a many-armed bandit—in fact, a bandit with a dynamically changing number of arms, where each arm represents a potential path of exploration toward an answer. (This is similar to the search through schemata in a genetic algorithm; see Holland, 1986.) There are far too many possible paths to do an exhaustive search, so in order to guarantee that in principle every path has a non-zero chance of being explored, paths have to be chosen and explored probabilistically. Each step in exploring a path is like sampling an arm, in that information is obtained that can be used to decide the rate at which that path should be sampled in the near future. The role of temperature is to cause the exponential increase in the speed at which promising paths are explored as contrasted with unpromising ones; as temperature decreases, the degree of randomness with which decisions are made decreases exponentially, so the speed at which good paths crowd out bad ones grows exponentially as more information is obtained. This type of strategy, in which information is used as it is obtained in order to bias probabilistic choices and thus to speed up convergence toward some resolution but never to absolutely rule out any path of exploration, is essential for flexibility in understanding and dealing with situations in the real world, in which there is a limited amount of time to explore an intractable number of possibilities.

Temperature affects the following decisions:

- The program's choice of which codelet to run next, based on relative urgency in the Coderack. At very high temperatures, this choice is fairly unbiased, meaning that all codelets on the Coderack have approximately an equal chance of being selected. As temperature falls, this choice becomes more and more biased, and at very low temperatures, the program is almost certain to choose one of the highest-urgency codelets next.

- A codelet's choice of which objects to use in scouting out or building a structure, based on salience. At high temperatures, all objects have roughly equal chance; at

low temperatures, the most salient objects are chosen almost all the time.

- A strength-tester codelet's decision whether to fizzle or to post a builder codelet, based on its calculation of the strength of the structure being considered. At high temperatures, strength is less strongly weighted in the decision.

- A builder codelet's decision whether or not to break already-existing incompatible structures, based on the competing structures' relative strengths. Again, at high temperatures, strength is less strongly weighted.

- A breaker codelet's decision whether or not to break a chosen structure, based on the structure's strength. Again, at high temperatures, strength is less strongly weighted.

- A codelet's decision of whether two nodes in the Slipnet are sufficiently close for the purpose of adding a new description to an object (such as adding the descriptor *first* to the **a** in **abc**) or making a slippage (such as *first* ⇒ *last*). At higher temperatures, the decision is made more randomly, and riskier (more distant) slippages have a better chance of being allowed.

- A rule-scout codelet's decision of which descriptors to choose for filling in the rule template, based on the descriptors' conceptual depth. At higher temperatures, conceptual depth is weighted less strongly.

- A group-scout codelet's decision whether or not to propose a single-letter group or to add a length description to a proposed group. At higher temperatures this decision is made more randomly, making the construction of single-letter groups and length descriptions—normally low-probability events—more likely.

The precise formulas for how temperature affects probabilistic biases are given in Appendix B.

Temperature allows Copycat to close in on a good solution quickly, once parts of it have been discovered. In addition, since high temperature means more randomness, temporarily raising the temperature gives Copycat a way to get out of ruts or to deal with snags; it can allow old structures to break and restructuring to occur so that a better solution can be found. That is, when the system runs into an impasse, the temperature can go up in spite of the fact that seemingly good organizing structures exist. Such a use of temperature is illustrated in the run of the program on "**abc** ⇒ **abd**, **xyz** ⇒ ?", given in Chapter 4.

Temperature in Copycat has some similarities to the notion of inhibitory control, discussed, for example, by Kaplan and Kaplan (1982). They note that when you are stuck in solving a problem, your inhibitory control gets lowered, making it easier for representations to emerge that were previously activated only below the level of consciousness. This makes it possible for you to come up with connections you didn't even know were there. However, when you lower inhibitory control, you run the risk of coming up with crazy, nonsensical ideas as well as useful new insights. High temperature corresponds to lowered inhibitory control. It allows structures that at low temperature would have been squelched immediately to be considered more seriously, and sometimes to be built. As will be seen in the next chapter, Copycat's use of temperature allows the program to come up with both insightful and bizarre solutions to certain problems (in particular, to the last three problems given in Section 2.1). The interesting thing is that the program has mechanisms that allow it to get reasonable and insightful solutions most of the time, while avoiding bad or crazy solutions fairly reliably (though it does get them from time to time).

3.7 Main Loop of the Program

At the beginning of a run of the program, the Coderack contains a standard initial population of codelets: an equal number of bottom-up bond-scouts, bottom-up replacement-finders, and bottom-up correspondence-scouts. In essence, the program assumes that these types of structures will be relevant in every problem. It might be wrong; for example, in the problem "**hjpb** ⇒ **xjpb**, **wlqzs** ⇒ ?", the letters were chosen randomly and there are no bonds to be found, but this fact would become clear only after some codelets had run.

The main loop of the program is as follows:

Until a rule has been built and translated, do the following:

Choose a codelet and remove it from the Coderack.

Run the chosen codelet.

If N codelets have run, then:

Update the Slipnet.

Post some bottom-up codelets.

Post some top-down codelets.

Finally, build the answer according to the translated rule.

Every N codelet-runs (where N is a parameter, currently set to 15), the Slipnet is

updated: for each node, any activation from instances discovered during the last N steps is added in, activation is spread between neighbors, and each node's activation decays at a rate determined by the node's conceptual depth.[5] In addition, various bottom-up and top-down codelets are placed on the Coderack. Top-down codelets are posted by active Slipnet nodes, but new bottom-up codelets are needed as well, not only because the initial set of bottom-up codelets might have missed certain possible structures, but also because new structures are being built and new objects (groups) are being created all the time, and very often these need to be themselves incorporated into higher-level structures. Relying on top-down codelets alone would often prevent the program from finding certain structures that didn't happen to correspond to previously active Slipnet nodes. So every N steps, not only top-down codelets but also bottom-up codelets of all the various types have some chance of being posted. (Some problems with determining the necessary number of codelets to post will be discussed in Chapter 6.)

Copycat has now been described in some detail, although, in order to strike a balance between completeness and clarity, certain less-central aspects of the program were left out of the discussion. Some of these will be given in Appendix B, which details the parameters and some of the formulas used in the system, and in Appendix C, which gives more detailed descriptions of the various codelet types.

The next chapter presents a statistical overview of Copycat's answers to the five target problems, and then follows the program through typical runs on each problem. This will give the reader a better idea of how all the pieces of the program fit together.

---

[5] Of course, this discrete updating process—every N steps—is meant to model the *continuous* activation, spreading activation, and activation decay that goes on in the mind. It could be made more continuous in the program by setting N to 1, for instance, but that would be computationally too expensive for the gain in continuity.

CHAPTER IV

COPYCAT'S PERFORMANCE ON THE FIVE TARGET PROBLEMS

4.1 Introduction

In this chapter I present the results of Copycat's performance on the five target problems discussed in Chapter 2. As was noted in that chapter, these problems were chosen because they illustrate, in an idealized and thus very clear form, some of the essential issues in high-level perception and analogy-making in general. In previous chapters I discussed a number of these issues and described the way in which the Copycat program models the mental mechanisms we are proposing in order to deal with these issues. Here I will present statistics summarizing what the program does on each of the five problems, and for each problem give a set of annotated screen dumps from one run (or in one case, two runs), which show how the mechanisms described in the previous chapter work together to produce the flexibility needed for the program to deal with a range of different situations in its microworld.

As was pointed out before, Copycat's abilities are not limited to these five problems alone, but rather, these problems represent something akin to a set of basis vectors defining a "vector space" of the program's abilities. In Chapter 5 I attempt to characterize this space by describing the program's performance on a set of *variations* of each of the five basic problems, where the variations explore how small changes in pressures affect the program's behavior.

Since the program is permeated with nondeterminism, different answers are possible on different runs. However, the nondeterministic decisions the program makes (e.g., which codelet to run next, which objects a codelet should choose, etc.) are all at a microscopic level, compared with the macroscopic level of what answer the program gets 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 "**abc ⇒ abd, ijk ⇒ ijl**", 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 "**abc ⇒ abd, ijk ⇒ ijd**". Thus, at a macroscopic level, the program is fairly deterministic: it gets the answer **ijl** almost all the time.

This notion of microscopic nondeterminism resulting in macroscopic determinism is often demonstrated in science museums using a contraption in which several thousand small steel balls tumble down through a dense grid of pins into one of many adjacent bins forming a horizontal row at the bottom. Though each ball takes a unique path at the microlevel, as more and more balls fall, the pattern of balls in the bins at the bottom gradually becomes a perfect gaussian curve, with most of the balls falling into the central bins, and fewer falling into the edge bins. In Copycat, the set of bins corresponds to the set of different possible answers, and the precise micro-path an individual ball takes corresponds to the actions of the program (at the level of individual codelets) during a single run. Given enough runs, a reliably repeatable pattern of answer frequencies will emerge.

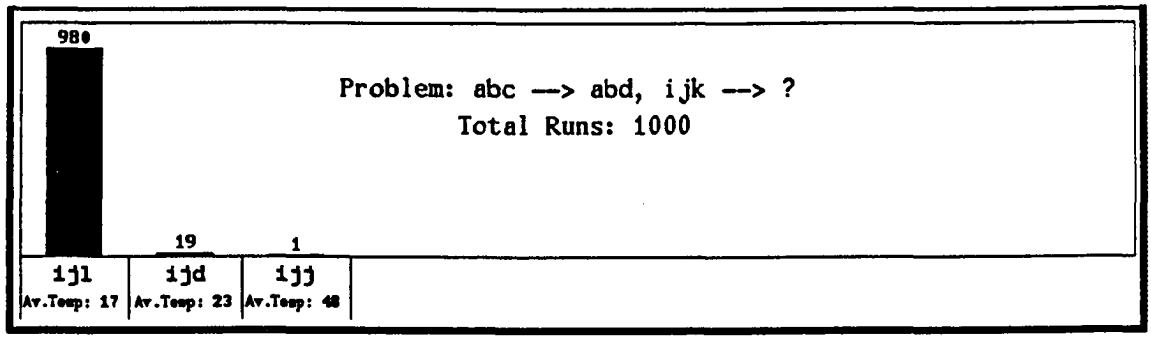
I present these patterns in the form of bar graphs, one for each problem, giving the frequency of occurrence and average end-of-run temperature for each different answer. For each of the five target problems, a bar graph is given, summarizing 1000 runs of Copycat on that problem. The number 1000 is somewhat arbitrary; after about 100–200 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** for "**abc ⇒ abd, ijk ⇒ ?**" (see bar graph below) begin to appear very occasionally; if 2000 runs were done on "**abc ⇒ abd, ijk ⇒ ?**", the program would give perhaps one or two other such answers, each once or twice. So even though 200 or so runs usually gives reliable statistics for the main range of answers to a given problem, I wanted to display at least a few of the fringe answers to each problem, so I ran each problem 1000 times. 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 (demonstrated by their appearance in the bar graphs), the mechanisms it has allow it to steer clear of them almost all of the time. As was mentioned before, the program (as well as people) has to have 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 be able to avoid following bad pathways, at least most of the time.

Along with each bar graph, I also give the results of a survey given to a number of undergraduate and graduate students at Indiana University, for the purpose of determining the range of answers people give on these problems. Ideally, Copycat should be able to get *all* of the answers that people get to a given problem—as long as those answers do not use knowledge that is not in the microworld—and it should never get answers that people find completely unjustified. This would indicate that the program is responding to the same pressures and perceiving the same things about the problems that people do (at least people adhering to the restrictions of the microworld). The frequencies and temperatures given here are not meant to be matched precisely with frequencies and preferences of answers given by people, since the program is not meant to model people at such a fine-grained level.

4.2 Frequency and Average Final Temperature of Answers for the Five Target Problems

The bar graph for "**abc ⇒ abd, ijk ⇒ ?**"



As can be seen, this bar graph summarizes 1000 runs of the program on "**abc ⇒ abd, ijk ⇒ ?**". 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 roughly corresponds to how *obvious* or *immediate* it 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 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 infrequency 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.) The average final temperature roughly corresponds to how *good* that answer seems to the program; the program assesses **ijl** (average temperature 17) to be somewhat better than **ijd** (average temperature 23), and much better than **ijj** (temperature 48).

One can get a sense of what the actual temperature values *mean* in terms of the quality of an answer by seeing how various sets of perceptual structures built by the program affect the temperature. This will be illustrated in detail in the next section. Roughly, an average final temperature below 30 indicates that the program was able to build a fairly strong, coherent set of structures—that it, in some sense, had a reasonable "understanding" of what was going on in the problem. Higher final temperatures usually indicate that some structures were weak, or that there was no coherent way of, say, mapping the initial string onto the target string. The program decides probabilistically when to stop and produce an answer, and though it is much more likely to stop when the temperature is low, it sometimes stops before it has had an opportunity to build all possible structures. For example, there are runs on "**abc** ⇒ **abd**, **ijk** ⇒ ?" on 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 **ijl** is about 7, which is about as low as the temperature ever gets.

There are also some problems 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. The only difference in the structures Copycat builds for these two answers is the rule: the former results from the rule "Replace rightmost letter by *D*", and the latter from the rule "Replace rightmost letter by successor". The latter rule is much more likely to be proposed (hence the higher frequency of **ijl**) and is also considerably stronger, but the problem is that the current formula for calculating the temperature (given in Appendix B) does not give enough weight to the strength of the rule. Thus, answers resulting from weak rules have lower final temperatures than they really deserve. This is a problem that should be addressed in future work on this project.

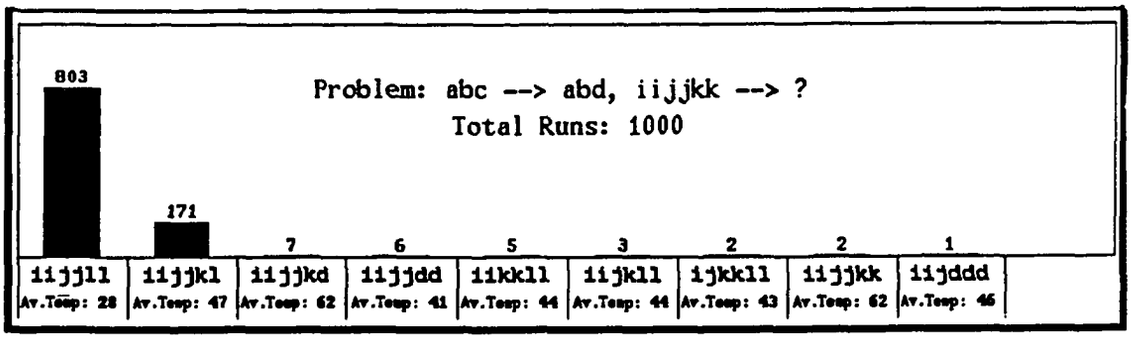
In the survey of people, 22 subjects answered this problem (after reading a description of the letter-string domain and its limitations). On all the problems, subjects were allowed to give multiple answers if they felt that there was more than one good answer, so the total number of answers for each problem is greater than the number of subjects, and many of the answers reflect the subjects' second, third, fourth, etc., choices (though on most problems, most people gave only one or two answers). The purpose of this survey was to collect *all* the different answers people, not just their preferred answers, so I have here listed all the responses I received. After each answer is listed the number of times it was given, though as I have said, the purpose of the survey was to compare Copycat and people's *range* of answers rather than the *frequencies* of different answers. The subjects were not asked to give justifications for their answers, but when reporting the results here, I will sometimes give what I presume the justification was.

For this problem, the answers people gave were:

1. **ijl** (21);
2. **ijk** (1);
3. **ijd** (1).

Copycat can get all three of these answers, although it did not get **ijk** during these 1000 runs. (Also, I am not sure what justification that subject had for that answer.)

The bar graph for "**abc ⇒ abd, iijjkk ⇒ ?**"



The bar graph above summarizes 1000 runs of Copycat on this problem. As can be seen, by far the most common (and lowest-temperature) answer is **iijjll**. The second most popular answer is **iijjkl**, which ignores the letter-groups in **iijjkk** and rigidly sticks to the rule of replacing the rightmost *letter* by its successor. After these two, all the other answers are very much on the fringes in terms of frequency, and none are considered to be of high quality. ("On the fringe" is a qualitative description of an answer, but it can be defined

roughly as an answer whose instances were produced on less than about 1 or 2 percent of the runs.) To get the answer **iijjkd**, the program describes the original change as "Replace rightmost letter by *D*", and follows it "to the letter". The next answer, **iijjdd**, reflects this same rule, but translated to take account of the group-structure of the target string (and since groups were noticed, the temperature is accordingly lower than for the previous answer).

The next several answers reflect various bizarre ways of viewing the target string. For **iikkll**, the program groups together the two rightmost *groups* in **iijjkk** (parsing the string as **ii-jjkk**), calling that larger group "the rightmost group", and replacing all the letters in it by their successors. The answer **iijkll** reflects a similar strange view, except the two **k**'s in **iijjkk** are grouped with only the rightmost **j** (**iij-jkk**), and these three letters are seen as "the rightmost group". The answer **iijjkk** comes from viewing the **abc ⇒ abd** change as "Replace *C* by *D*", and since the target string has no instances of *C*, it is left alone. Finally, the answer **iijddd** is similar to the answer **iijkll**, but involves replacing the "rightmost group" of three letters by **d**'s rather than by successors. The reasons Copycat came up with some of these strange answers will be analyzed further in Chapter 6, which discusses some problems with Copycat. Happily, these last five answers account for only 1.3% of the total, and perhaps more significantly, none has a final temperature lower than 41, which in Copycat's terms is fairly high; the program considers **iijjll**, with an average final temperature of 28, to be much more reasonable.

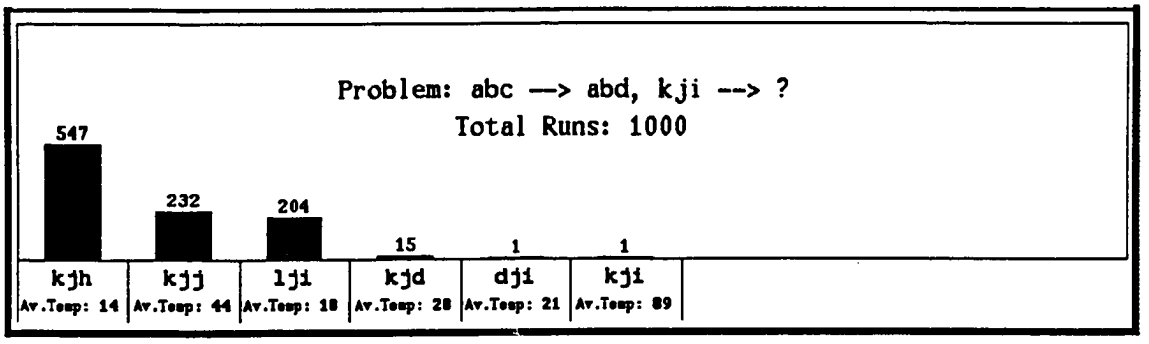
In the survey of people, 18 subjects answered this problem. The answers were:

1. **iijjll** (13);
2. **iijjkl** (8);
3. **iikjkl** (5) (replace the rightmost letter of each group of three by its successor);
4. **iikkll** (2) (replace all letters after the leftmost two by their successors);
5. **iijjkd** (1);
6. **iijjdd** (1);
7. **iikjkk** (1) (replace the third letter by its successor).

Copycat can get answers 1, 2, 5, and 6 (it gets answer 4, but not for the same reason people do). It cannot get the other answers given since it lacks the concept of "third letter", and cannot make the descriptions "leftmost two letters" and "rightmost letter of a group". As can be seen from the bar graph for this problem given above, Copycat produces (though

rarely) a number of answers that people never give (e.g., **iijkll**), based on bad groupings of the target string (i.e., groupings that people would judge to be completely unmotivated).

The bar graph for "**abc ⇒ abd, kji ⇒ ?**"



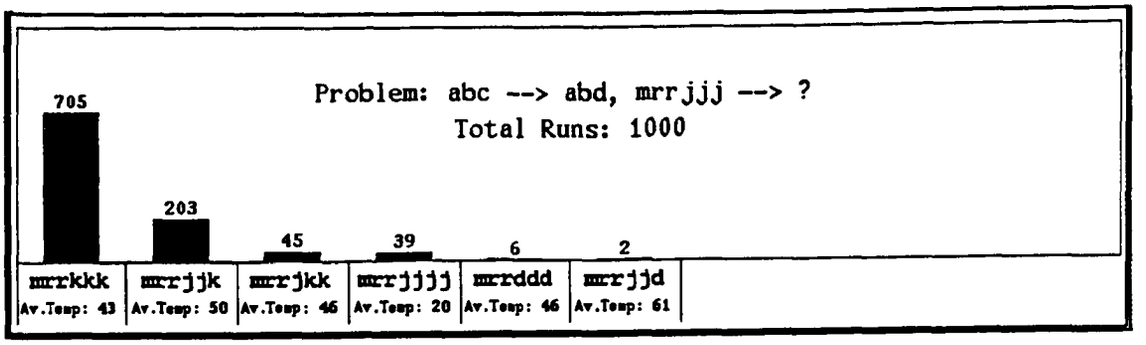
The bar graph above summarizes 1000 runs of Copycat on this problem. As can be seen, there are three answers that predominate, **kjh** being the most common (and having the lowest average final temperature), with **kjj** and **lji** almost tying for second (the latter being a bit less common, but having a much lower average final temperature). The answer **kjd** comes in a very distant fourth, and there are two "fringe" answers with only one instance of each: **dji** (a mixture 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**—notice the relatively low temperature on this answer, indicating that a strong set of structures was built!), and **kji** (again reflecting the literal-minded rule "Replace *C* by *D*"), which has a very high temperature of 89, indicating that on this run almost no structures were built before the program decided to stop.

In the survey, 10 subjects answered this problem. The answers were:

1. **kjj** (8);
2. **kjh** (3);
3. **lji** (1);
4. **kjd** (1);
5. **kjl** (1) (this is either a confusion of answers 2 and 3, or an insistence on changing the rightmost letter, even though the subject thought it should be changed to the letter after the latest in alphabetical order).

Again, the range rather then the frequency of the different answers is the point of comparison here. Copycat gets all of these except the last, somewhat confused answer. It also gets (on rare occasions) **dji** and **kji**, which no one in the survey gave.

The bar graph for "abc ⇒ abd, mrrjjj ⇒ ?"



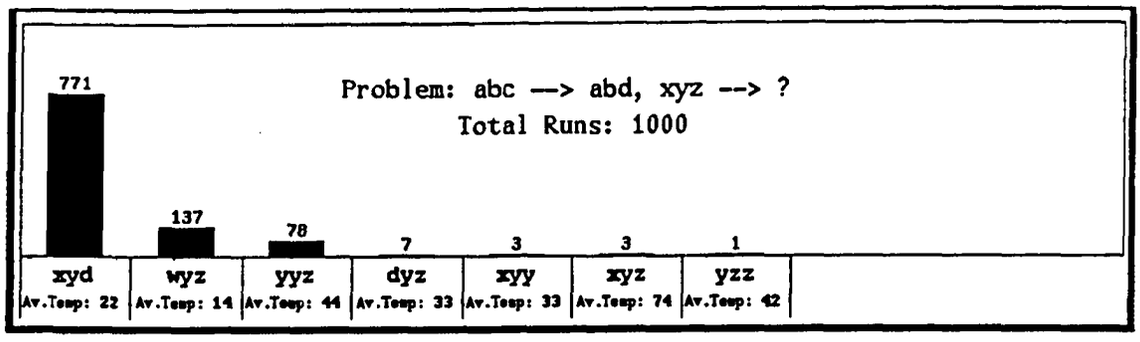
The bar graph above summarizes 1000 runs of Copycat on this problem. As can be seen, the most common answer by far is the straightforward **mrrkkk**, with **mrrjjk** a fairly distant second. For Copycat, these are the two most immediate answers; however, the average final temperatures associated with them are fairly high, because (as was discussed in Chapter 2) of the lack of any coherent structure tying together the target string as a whole. Next there are 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**, which was discussed in Chapter 2. The average final 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 of a solution is by no means always perfectly correlated with its quality. The other two answers, **mrrddd** and **mrrjjd**, come from replacing either a letter or a group with **d**'s, and are on the fringes.

In the survey, 34 subjects answered this problem. The answers were:

1. **mrrkkk** (19);
2. **mrrjjk** (12);
3. **mrsjjk** (4) (a result of parsing the string as **mrr-jjj**, and replacing the third letter of each group by its successor);
4. **mrsjjj** (2) (replace the third letter by its successor);
5. **mrrjkk** (2) (a result of parsing the string as **mr-rj-jj** to correspond to **a-b-c**, and replacing the rightmost group by its successor);
6. **mrskkk** (1) (replace everything following leftmost two letters by its successor);
7. **mrrjjj** (1) (I am not sure why the subject gave this answer);
8. **mrrjjjj** (1).

Copycat is able to get 1, 2, 7, and 8. (Copycat also gets 5, but the program's reason is completely different. I am not sure Copycat gets 7 for the same reason the subject gave that answer). The lack of concepts such as "third letter" and "leftmost two letters", and the inability to group letters that are not related (such as **m** and **r**, which were grouped for answer 5) prevent the program from getting the other answers. There are three answers Copycat got in the 1000 runs displayed above that none of the subjects in this survey got: **mrrddd**, **mrrjjd**, and **mrrjkk** (identical to answer 5, but not given for the same reason). I think if the survey were larger, the first two of these answers would show up, but I think that it is very unlikely that a person would group the rightmost two **k**'s for no good reason, as Copycat does.

The bar graph for "**abc ⇒ abd, xyz ⇒ ?**"



The bar graph above summarizes 1000 runs of Copycat on this problem. As was discussed earlier, the answer **xya** is not available to the program; by design, *Z* has no successor. On 98% of the runs, the program tries to take the successor of *Z* and fails, which then forces it to do some restructuring (and, as will be seen in the screen dumps later in this chapter, Copycat often hits the same snag again and again in the same run—on average 9 times per run—before it succeeds in finding a way of solving the problem). As can be seen, 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**. A distant second in frequency, but the answer with the lowest average temperature, is **wyz**, which many people (including myself) consider to be the most elegant solution. To get this answer, the program has to restructure its perceptions of what corresponds to what, noticing that *A* and *Z* are at opposite ends of the alphabet, so there is a plausible correspondence between them if the spatial *and* alphabetic directions of the two strings (**abc** and **xyz**) are also seen as opposite. The next answer, **yyz**, reflects a view that neglects the opposite alphabetic direction of the two strings, and although it allows the *leftmost* letter to be replaced, it

insists on holding fast to the notion of replacing it by its *successor*, since the rightmost letter of **abc** was replaced by *its* successor.

The other four answers are on the fringes as far as frequency goes. The answer **dyz** (like **dji** in "**abc** ⇒ **abd**, **kji** ⇒ ?" above) is a comical blend of intelligence and rigidity; it exhibits a good deal of flexibility in the willingness to slip *rightmost* to *leftmost*, but it holds a rigid view of the **abc** ⇒ **abd** change. (This contradictory mixture of intelligence and rigidity is very much akin to the notion of *frame blends* described earlier. Many people find this answer funny, and indeed, frame blends are central in certain kinds of humor. Some connections of such answers in the letter-string domain with frame blends and jokes are discussed in Hofstadter & Gabora, 1990.) The answer **xyy** allows that the two strings are to be perceived in opposite *alphabetic* directions (thus a *successor* ⇒ *predecessor* slippage), but refuses to give up the idea that the strings have the same *spatial* direction, and thus insists on changing the *rightmost* letter, as was done in **abc**. The answer **xyz** comes from reinterpreting the **abc** ⇒ **abd** change as "Replace *C* by *D*"; and finally, the answer **yzz** is a strange variant of **yyz**, in which the **x** and **y** in **xyz** are grouped together as one object, which is then replaced as a whole by its "successor" (the successor of each letter in the group).

In the survey, 34 subjects answered this problem. The subjects were allowed to answer **xya** (and virtually all of them did so) but then they were informed that, since Copycat doesn't have the concept of circularity, it cannot produce this answer. They were then asked to come up with one or more different answers.

There were a large number of different answers given:

1. **xyz** (9) (if the **z** can't be changed, then just leave it alone);
2. **xyy** (8) (if the **z** can't be moved forwards in the alphabet, then the next best thing is to move it backwards);
3. **xy** (5) (if **z** has no successor, then it just falls off the end of the string);
4. **xyd** (5);
5. **wxz** (4) (based on the desire to imitate the alphabetic space between the two rightmost letters in **abd**, which can be done by moving the leftmost two letters backwards in the alphabet);
6. **xzz** (1) (if you can't take the successor of the **Z**, the next best thing is to take the successor of its neighbor, the **y**);
7. **wyz** (1);

8. **xyw** (1) (using similar reasoning to that which yields **wyz**, but insisting that the rightmost letter be the one that is replaced);
9. **zxw** (1) (the subject specified that they wanted to both change the rightmost letter and at the same time imitate the relationships in **abd**; by reversing the string and then replacing the rightmost letter by its predecessor, they were able to do this);
10. **xyx** (1) (the subject specified that they wanted to answer **xyy**, but didn't like the fact of the double **y**, so instead used the letter before *Y*);
11. **abd** (1) (replace the whole string by **abd**).

Copycat is able to get answers 1 and 2 (though possibly not for exactly the same reasons that these subjects gave them) as well as 4 and 7. The other answers involve concepts or operations that Copycat is incapable of (such as dropping a letter, as in **xy**). Some of these answers (like some of the answers given by Copycat) seem to be frame blends, where the person perceives a flexible way of answering, but insists on rigidly holding on to some aspect of the initial **abc** ⇒ **abd** change (such as insisting that the *rightmost* letter must be changed). In the 1000 runs on this problem displayed earlier, Copycat did get some answers that weren't given by any of the subjects in this survey: **yyz** (which people have given from time to time in more informal surveys), **dyz** (which people never give, except jokingly), and **yzz** (a "bad grouping" answer).

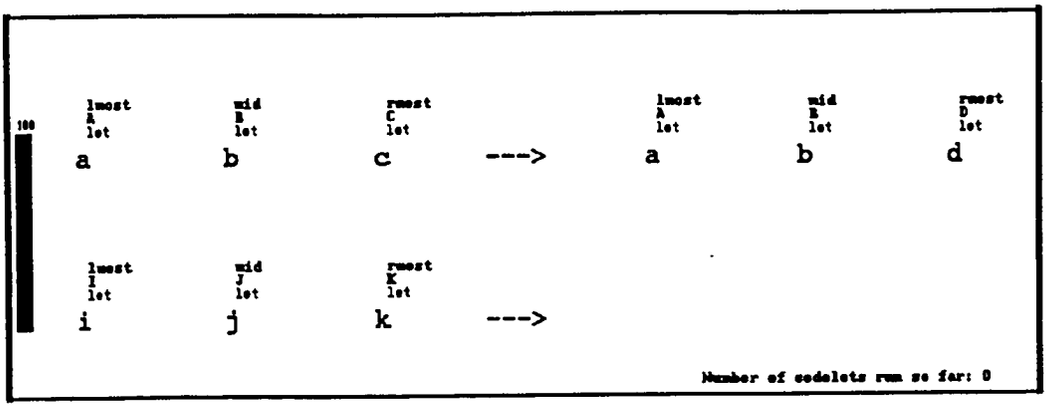
This section has given statistics for Copycat's performance on the five target problems, and compared the range of answers given by the program with that of people. The same sorts of statistics and comparisons will be given for a larger set of problems (all variants of the five target problems) in the next chapter, along with a summary of all the comparisons and a discussion of the overall performance of the program with respect to the artificial-intelligence and psychological criteria proposed earlier.

4.3 Screen Dumps from Runs on the Five Target Problems

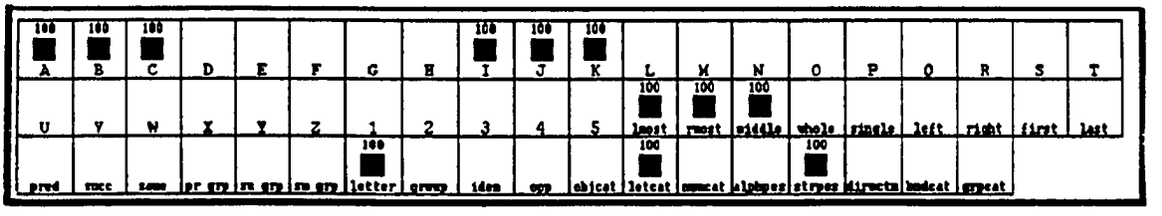
In this section, annotated screen dumps of Copycat's graphics are given for runs on each of the five problems. These screen dumps are meant to make clearer how the program actually solves these problems. On each run, the Workspace is displayed, and on some runs the Slipnet is displayed as well.

abc ⇒ abd, ijk ⇒ ?

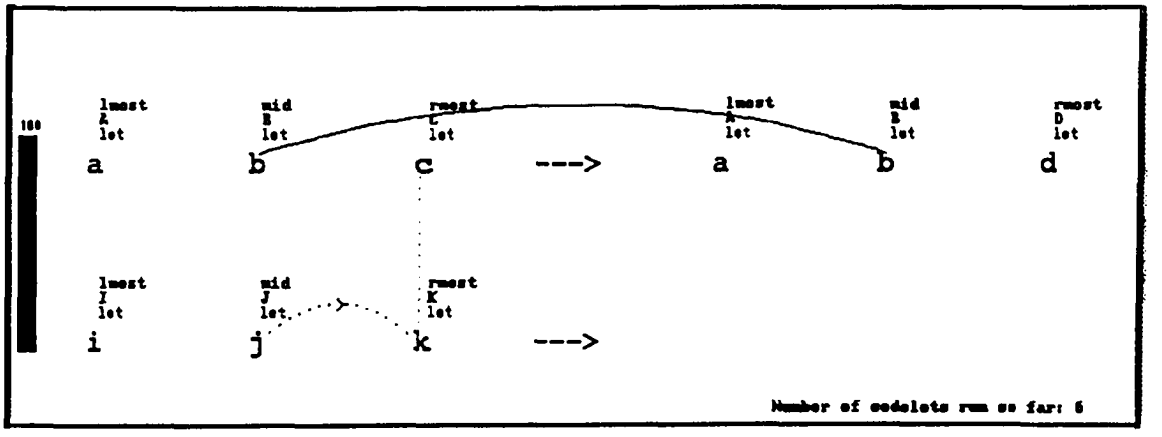
The following is a set of screen dumps from a fairly typical run of Copycat on this problem.



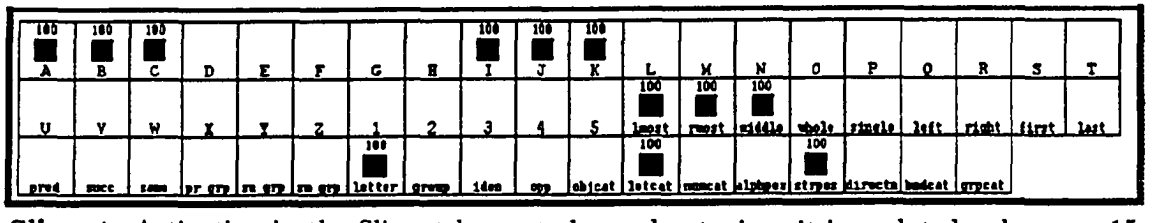
1. **Workspace:** The program is presented with the three strings in the Workspace. Each letter has a list of initial descriptions, including *string-position*, *letter-category*, and *object-category* descriptions (the first two description-types are initially relevant by default; relevant descriptions appear in boldface). The temperature, represented by a "thermometer" at the left, is at its maximum value of 100 degrees (0 is the minimum), so initial decisions are made fairly randomly, though there are still some biases, even at the highest temperature. This screen dump was made before the run began; as is indicated at the bottom right of the Workspace, no codelets have run so far. .



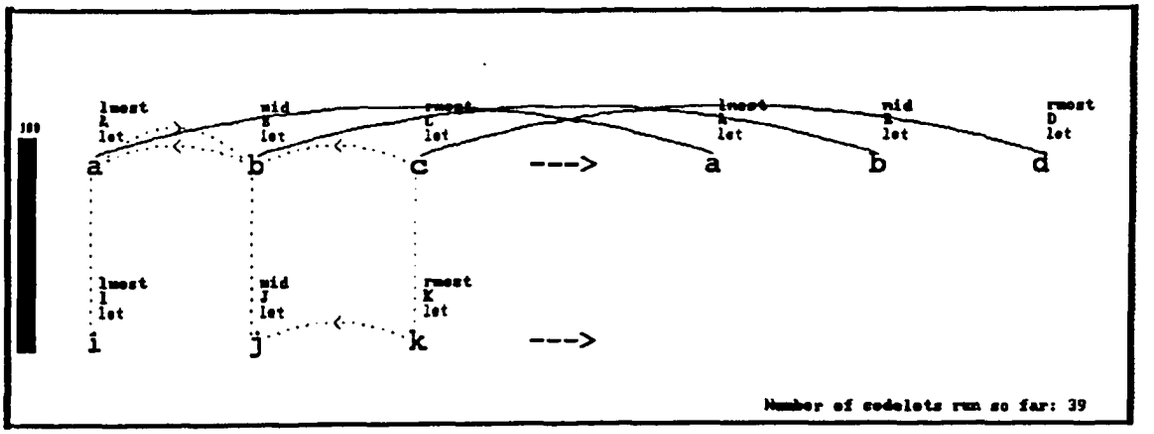
Slipnet: The Slipnet is displayed above (nodes only; no links are shown). The black square inside any node's rectangle represents its activation: the size of the square is proportional to the level of activation, and the actual numerical level, ranging from 0 to 100%, is displayed above each square. The nodes are (in the order displayed): *A–Z*, *leftmost*, *rightmost*, *middle*, *whole*, *single*, *left*, *right*, *first*, *last*, *predecessor*, *successor*, *sameness*, *predecessor group* (abbreviated "pr grp"), *successor group* (abbreviated "su grp"), *sameness group* (abbrieviated "sm grp"), *letter*, *group*, *identity*, *opposite*, *object-category*, *letter-category*, *number-category*, *alphabetic-position*, *string-position* (abbreviated "strpos"), *direction*, *bond-category* (abbreviated "bndcat"), and *group-category* (abbreviated "grpcat"). As can be seen, the nodes corresponding to the initial descriptions given to each letter are activated, each at 100%.



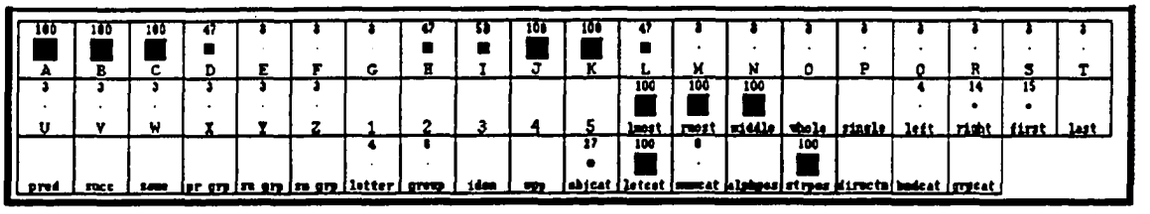
2. **Workspace:** Six codelets have run, and a few have had some success. The modified-string replacement for the **b** in **abc** has been found (solid arc across the arrow), a bond has been proposed in the target string from the **j** to the **k** (dotted arc), and a correspondence has been proposed between the **c** and the **k** (dotted vertical line). In general, structures that have been *proposed* are represented by dotted or dashed arcs and lines, and structures that have been *built* are represented by solid arcs and lines. Since no initial-string or target-string structures have been actually built yet, the temperature remains at 100 (the initial-string–modified-string replacement arcs do not affect the temperature).



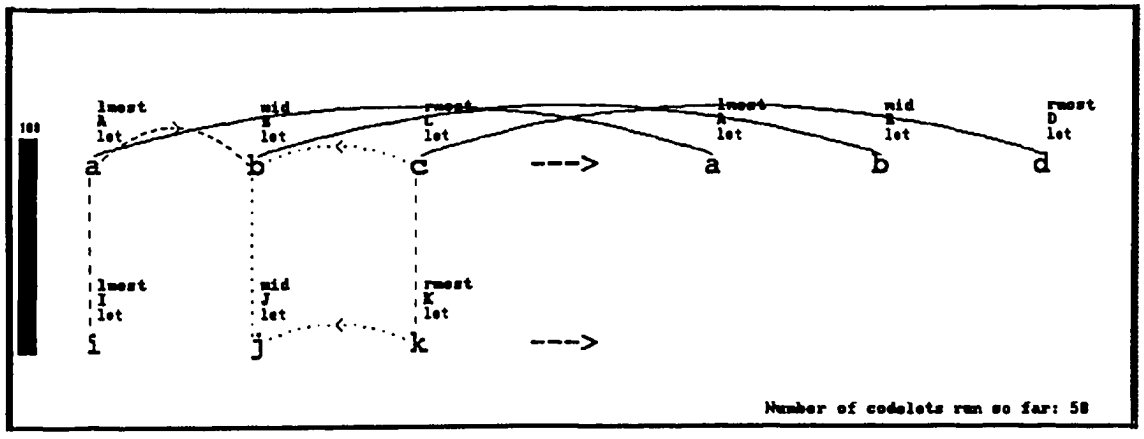
Slipnet: Activation in the Slipnet has not changed yet, since it is updated only every 15 codelets.



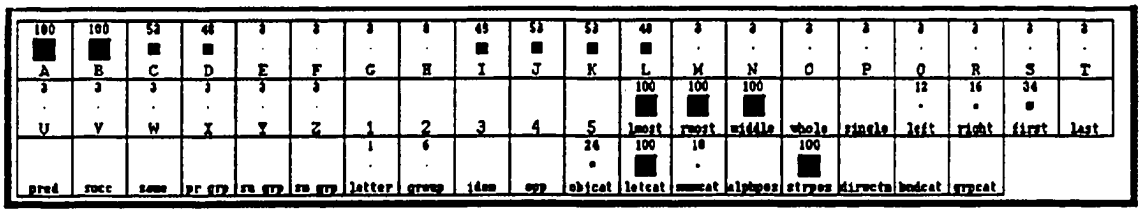
3. **Workspace:** Now 39 codelets have run. All three initial-string–modified-string replacement arcs have been built (since determining these is trivial, they are almost always constructed quite early on). Several possible bonds and correspondences are being considered, but since none has been built yet, the temperature remains at 100.



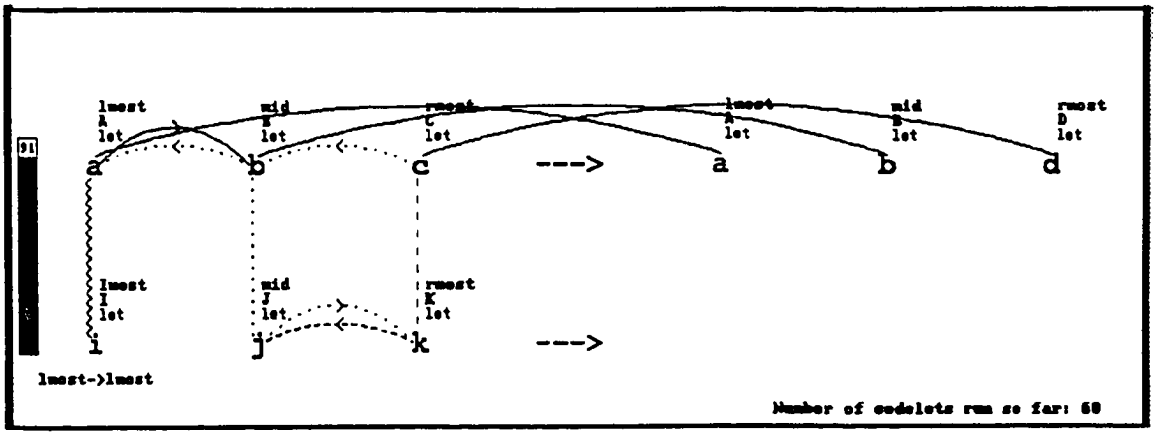
Slipnet: The initial activations have decayed and spread in various ways (e.g., the node *letter*'s activation has decayed, all 26 letter-category nodes have received a tiny bit of activation from the node *letter-category*, and *letter-category* has also spread some activation to *object-category* and *number-category*), and additional activation has come from codelet actions in the Workspace (e.g., the letter-categories involved in the proposed bonds have been reactivated).



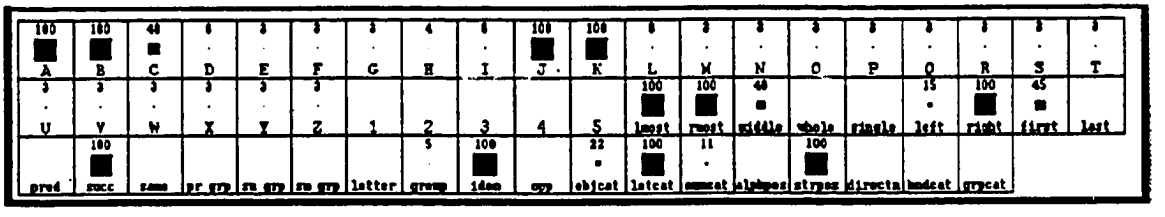
4. **Workspace**: 50 codelets have run, and various proposed structures can be seen at various stages of consideration. The *dotted* arcs and lines (the proposed **c**–**b** and **k**–**j** predecessor bond ands the proposed **b**–**j** correspondence) are structures that have been proposed by a scout codelet and are waiting to be examined by a strength-tester codelet. The *dashed* arcs and lines (the proposed **a**–**b** successor bond and the proposed **a**–**i** and **c**–**k** correspondences) are structures that have passed their respective strength-tester's evaluation and are waiting to be built by a builder codelet. Note that there are many actions not shown in these screen dumps, e.g., the actions of scout codelets that fizzle without proposing anything (this would happen if, say, a correspondence-scout codelet examined the **c** and the **j** to see if there were any grounds for a correspondence between the two; there wouldn't be any).



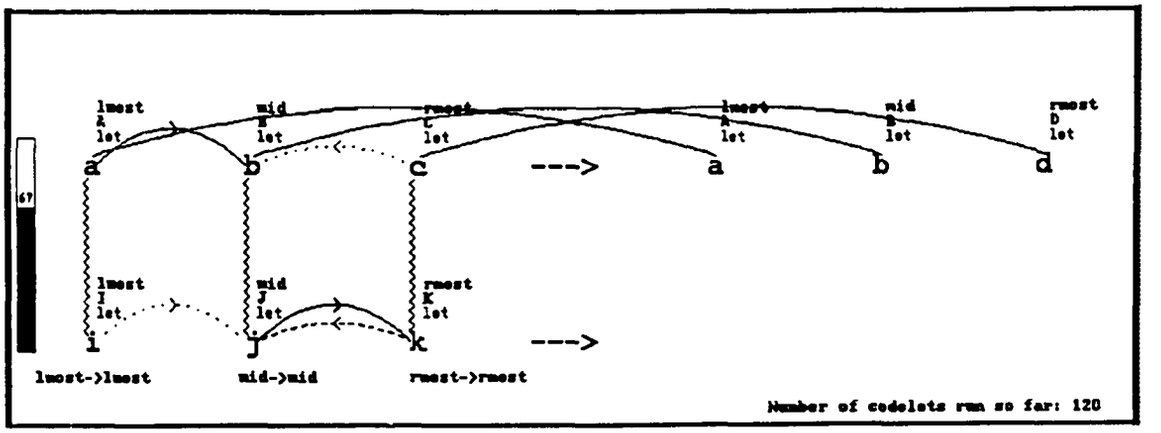
Slipnet: Further activation spread and decay has occurred (e.g., *first* has received some activation from *A*), and nodes whose instances have been recently examined by codelets (e.g., *A* and *B*) have received additional activation.



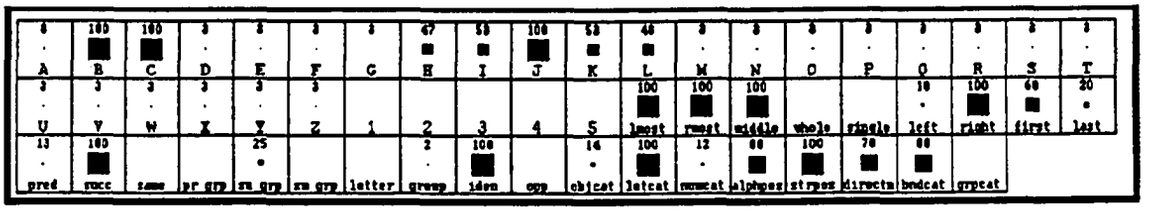
5. **Workspace**: After 60 codelets have run, some structures have finally been built, and appear as *solid* arcs or jagged lines: the a–b successor bond and the a–i correspondence. Beneath the correspondence is its single concept-mapping: *leftmost* ⇒ *leftmost*. In response to these structures, temperature has fallen to 91. Other proposed structures are still in the process of being explored at different spatial locations and at different speeds.



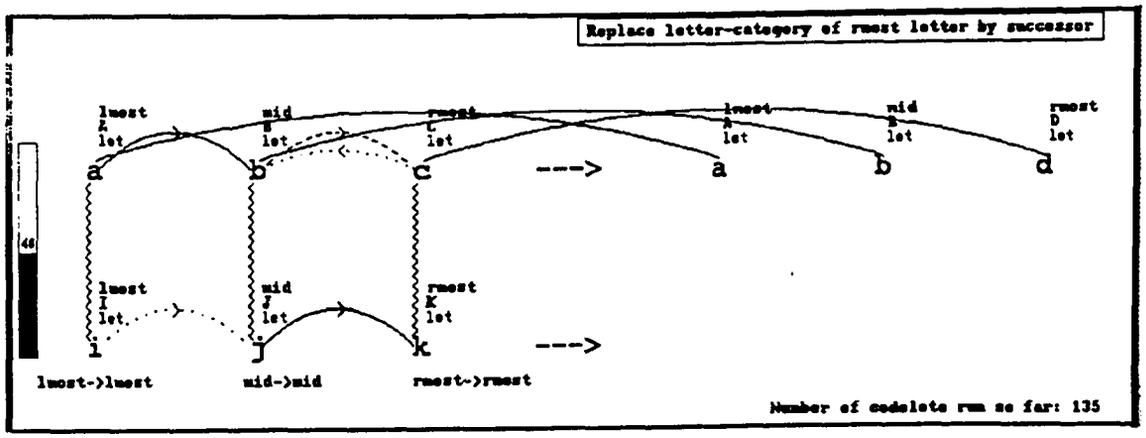
Slipnet: The newly built structures have affected the Slipnet: the nodes *successor* and *right* (corresponding to the category and direction of the bond) are activated, as is the node *identity* (corresponding to the type of concept-mapping underlying the a–i correspondence).



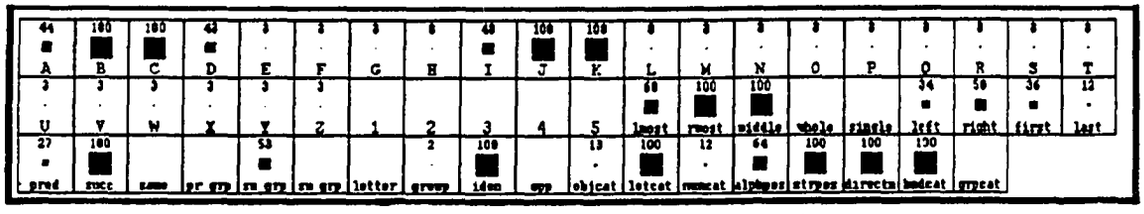
6. **Workspace:** The active nodes *successor* and *right* have begun to exert a top-down influence in two ways: by increasing the strength of proposed bonds of these types (the activation of *successor* causes successor links in the Slipnet to shrink, making the bond between, say, *I* and *J* stronger) and by causing codelets to be posted expressly to look for such bonds. Another such bond has been built between the **j** and the **k**, and an **i–j** bond is being considered. Also, correspondences have been built from all three letters in the initial string to letters in the target string, based on their string-position descriptions. In response to these structures, the temperature has fallen to 67.



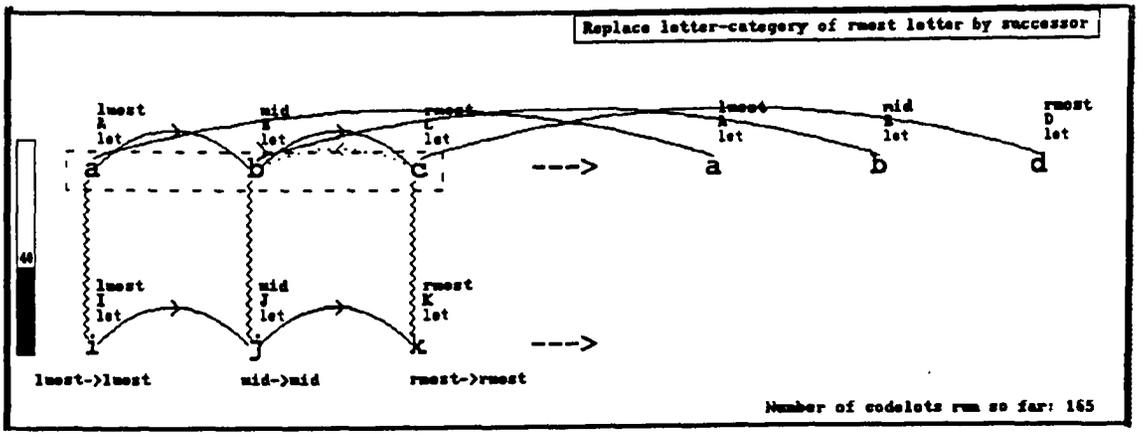
Slipnet: The node *first* is becoming active, due to continuing activation of *A*. The node *first* has also spread activation to *last* as well as to *alphabetic position*, which will post codelets to try to make such descriptions. (As will be seen, *alphabetic-position* will decay, and won't have much effect in this problem.) Likewise, *successor* and *right* have spread small amounts of activation (respectively) to *predecessor* and *left*, but not enough for these to have any influence yet. *Successor* and *right* have also spread activation (respectively) to the more general nodes *bond-category* and *direction*.



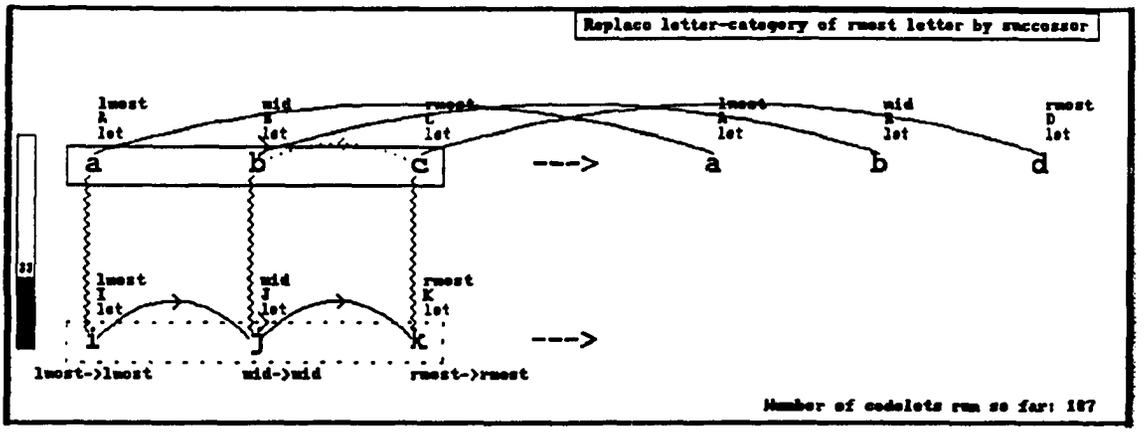
7. **Workspace:** A coherent view, based on right-going successor bonds, is beginning to emerge, enforced by the presence of such bonds in the two strings as well as the top-down influence from the active nodes in the Slipnet. Some of the explorations along different lines, seen in the previous screen dumps, are proceeding relatively slowly (e.g., the proposed **c–b** predecessor bond, which is still waiting for its strength-tester to run) or have fizzled entirely (e.g., the previously displayed proposed **k–j** predecessor bond) in response to these pressures. A rule ("Replace letter-category of rightmost letter by successor") has been constructed to describe the **abc–abd** change; it appears in a box above the modified string. The temperature has fallen to 48 in response to the building of this strong rule.



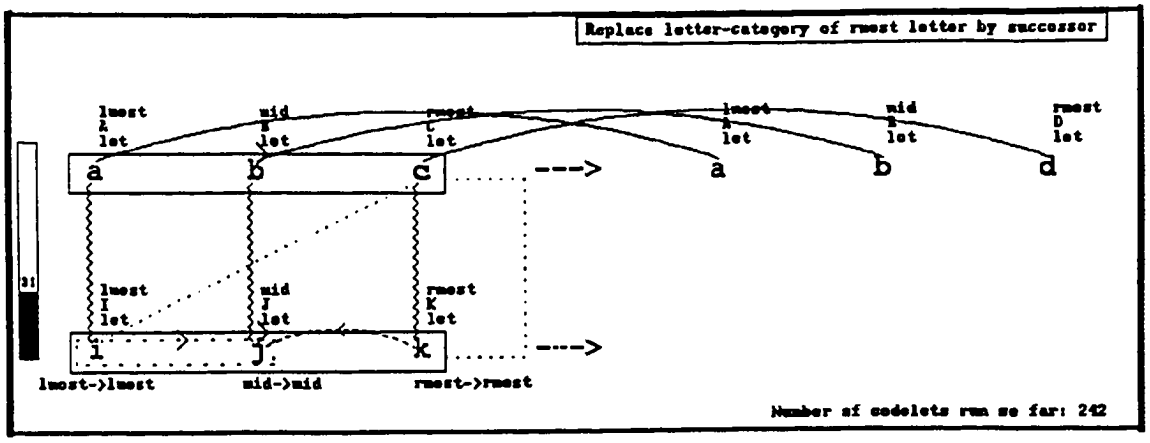
Slipnet: *Successor* has spread activation to *successor-group* (abbreviated as "su grp"), which is now active enough to begin posting codelets to look for such groups. The nodes *first* and *alphabetic-position* have lost some activation through decay.



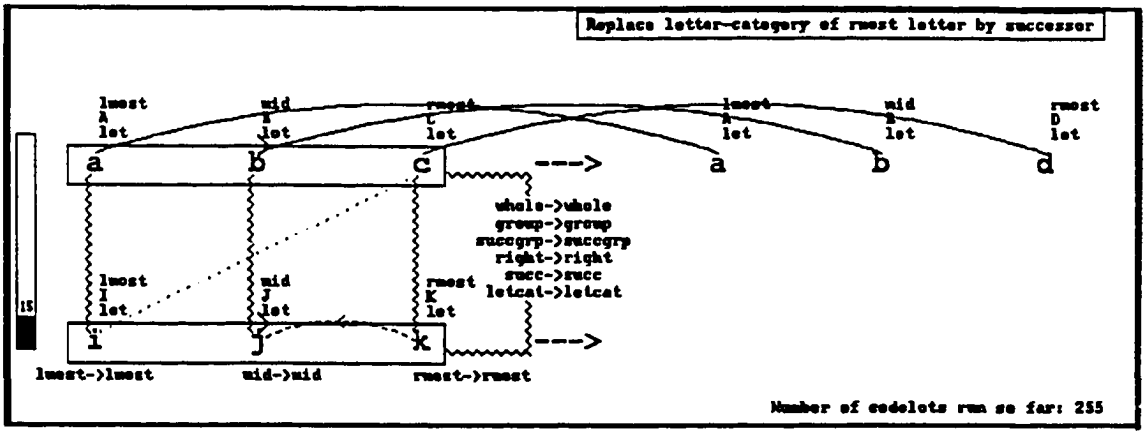
8. **Workspace:** Now, 165 codelets into the run, the notion of right-going successor bonds has taken over almost completely (though other possibilities are still being explored, albeit much more slowly—e.g., the proposed **c–b** predecessor bond) and a grouping of the initial string as a whole is being considered. The temperature has fallen to 40, reflecting the program's assessment of the promise of the structures it has built so far. As the temperature gets lower and lower, the program's decisions become more and more deterministic, its behavior more and more serial (i.e., a small number of high-urgency codelets overwhelm a larger number of low-urgency ones, so fewer and fewer other possibilities are being considered), and its actions more and more dominated by top-down forces (as top-down codelets crowd out bottom-up ones). A single dominating point of view begins to be "frozen" into place. (The state of the Slipnet is similar here to its state in the previous frame.)



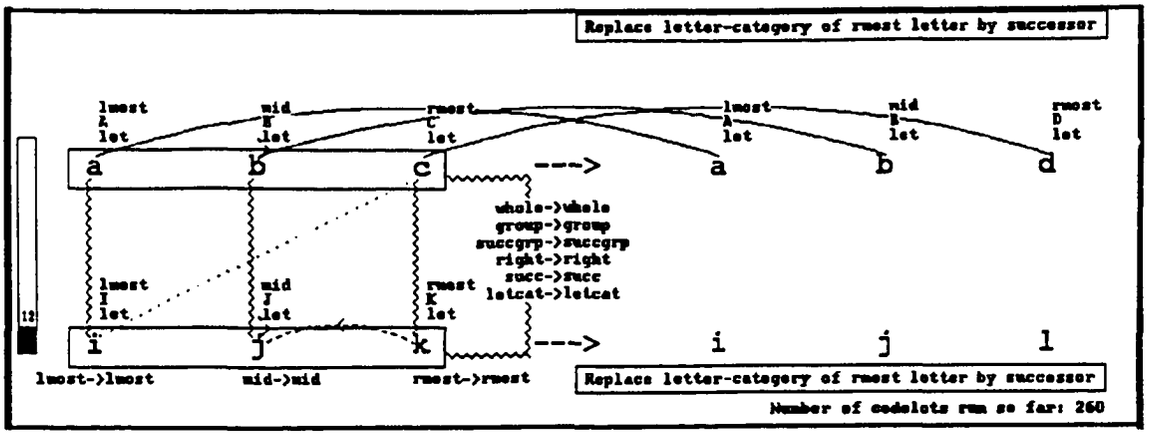
9. **Workspace:** The entire initial string is now being seen as a right-going successor group (the direction is represented by a right-going arrow at the top of the rectangle; the bonds inside the group still exist, but their display has been suppressed). This creates pressure to view the target string in the same way, and indeed a similar grouping is being considered there.



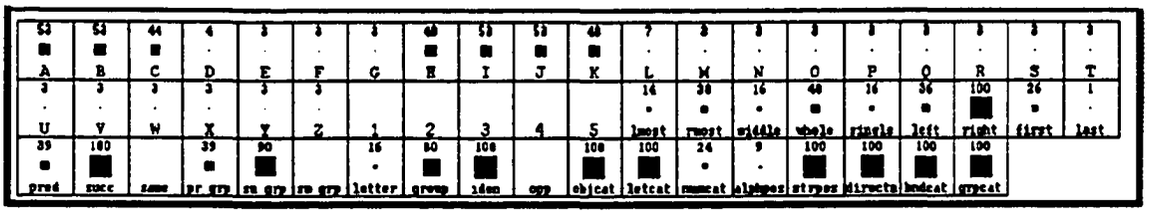
10. **Workspace:** Both strings have now been grouped as wholes, and a correspondence is being considered between the two groups (dotted bent line to the right of the two groups). The temperature has fallen to 31, but even at this relatively late stage, a few other rival possibilities are still being explored (though with very low urgency): a left-going k–j predecessor bond, a group in the target string containing only the i and the j, and a correspondence between the c and the i, based on the (here) fairly weak link between *rightmost* and *leftmost* in the Slipnet. None of these structures (especially the last) are very strong, and given the strong and coherent set of structures that have been built, these rivals have very little chance of getting anywhere at this point.



11. **Workspace:** The group–group correspondence has been built with all its concept-mappings listed alongside it (*letcat* ⇒ *letcat* means that both groups are based on bonds between *letter-categories*). The temperature has fallen to 15, indicating the program's assessment that it is very close to a good answer.



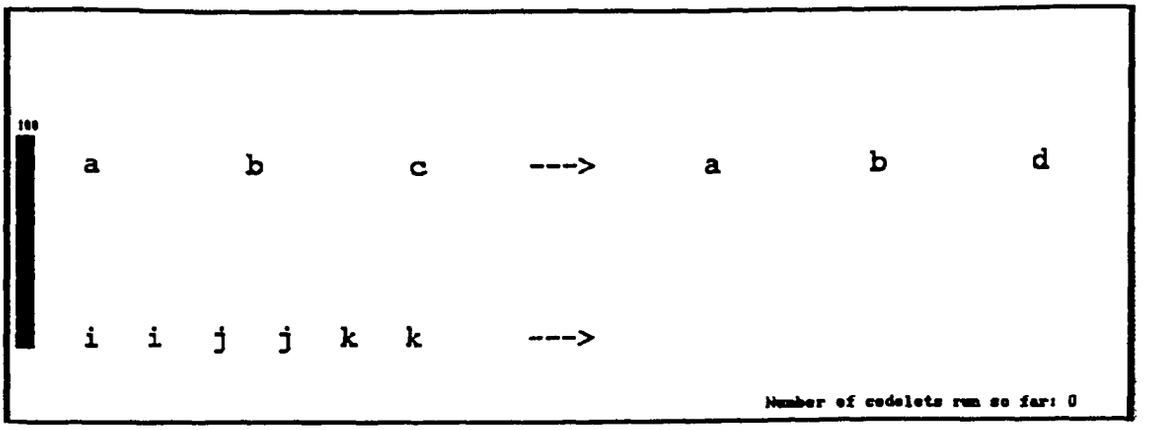
12. **Workspace:** The rule has been "translated" (although since all the concept-mappings are identities, no changes needed to be made) and the answer **ijl** has been constructed according to the translated rule (the answer, with the translated rule beneath it, appears at the right-hand side of the screen). The final low temperature of 12 indicates that the program is very satisfied with this answer. This run consisted of 260 codelets (the average number of codelet steps for this problem is about 290 codelets).



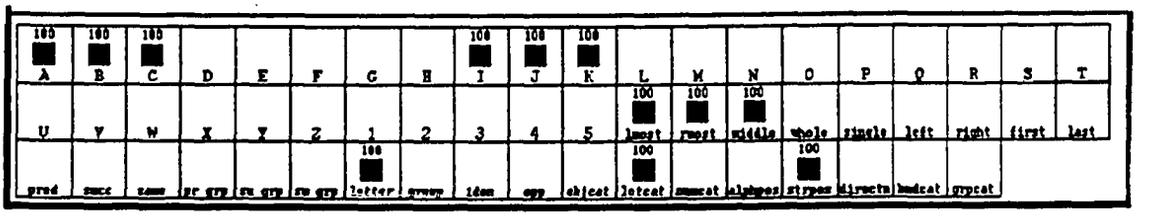
Slipnet: The final configuration of the Slipnet indicates what concepts were found to be relevant in this problem: the individual letter-categories' activations have decayed, and the notions of *right*, *successor*, *successor-group*, *group*, and *identity* are activated, along with nodes corresponding to various categories (e.g., *bond-category*).

abc ⇒ abd, iijjkk ⇒ ?

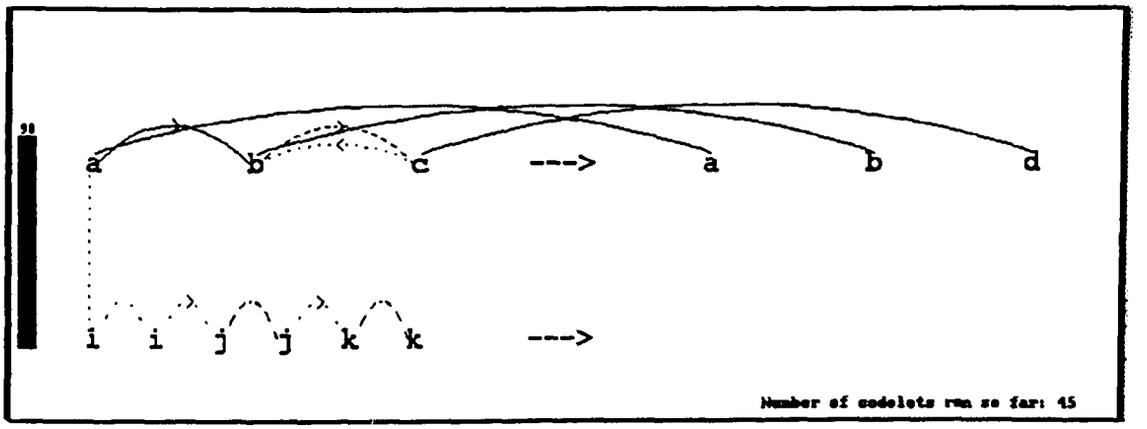
The following is a set of screen dumps from a fairly typical run of Copycat on this problem. The Slipnet is displayed only on frames where there is a point to be made about it.



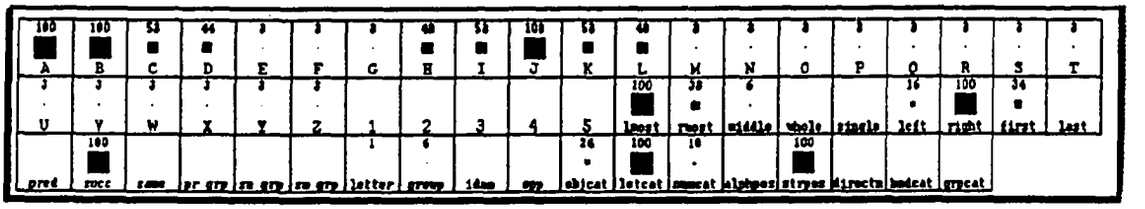
1. **Workspace:** The program is presented with the three strings. For the sake of clarity, the descriptions of each letter are not displayed here, but they are as given in Figure 3.4.



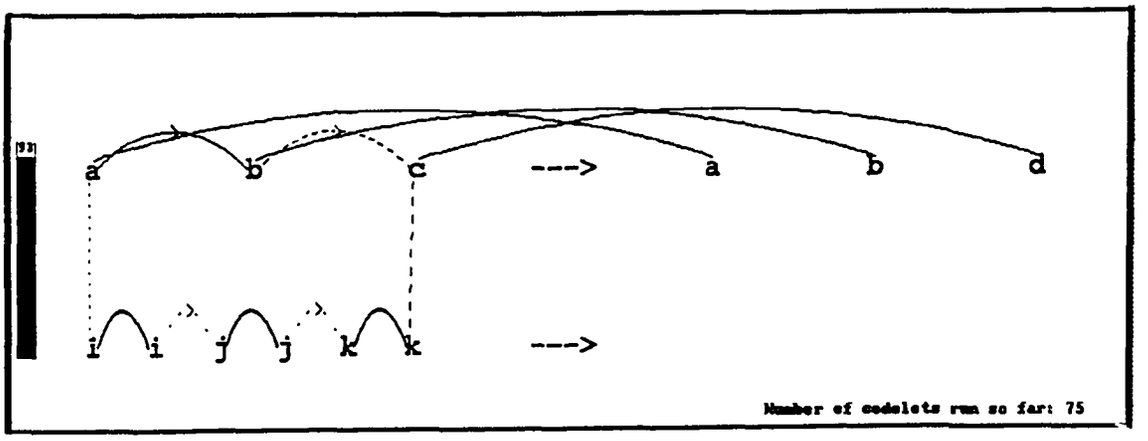
Slipnet: The Slipnet starts out in the same initial state as in the last problem.



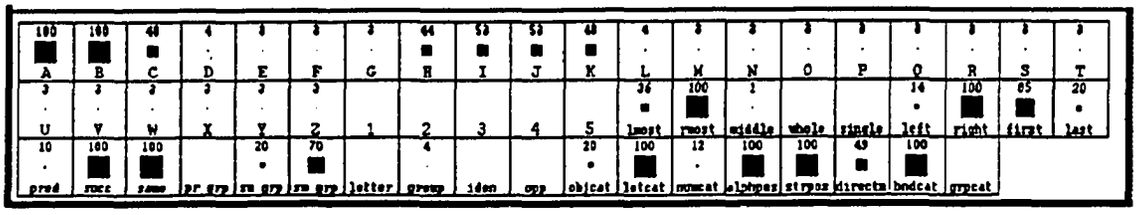
2. **Workspace:** After 45 codelets have run, an **a–b** successor bond has been built, and several possible structures are being considered. Sameness bonds (e.g., between the two **k**'s) are intrinsically stronger than successor and predecessor bonds, so they tend to be evaluated and built more quickly (their codelets receive higher urgencies).



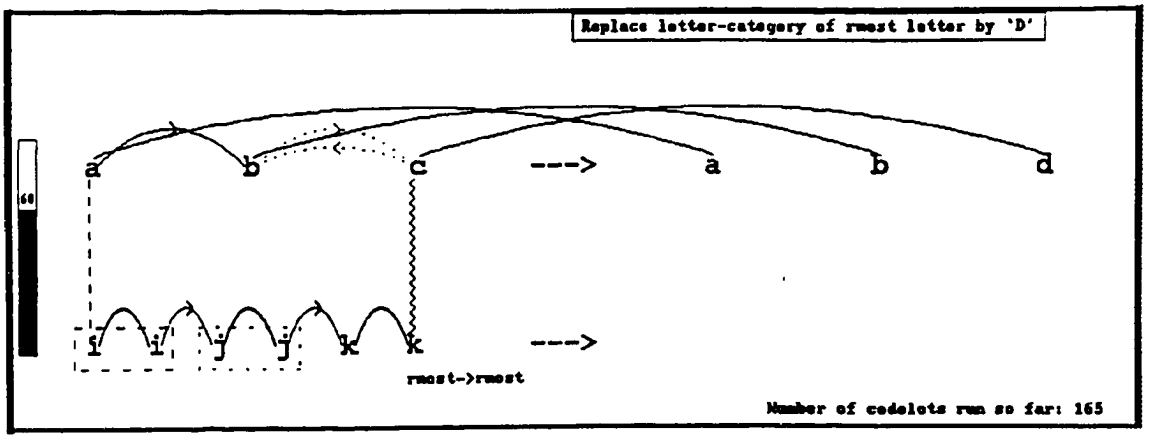
Slipnet: Activation has spread and decayed from the initially active nodes, and some other nodes have been activated in response to actions of codelets in the Workspace: for example, the nodes *successor* and *right* have been activated in response to the bond that was built.



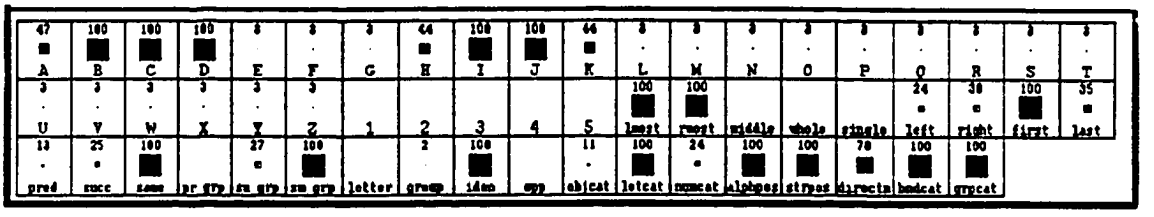
3. **Workspace:** Three sameness bonds have been built in the target string, and other structures continue to be considered.



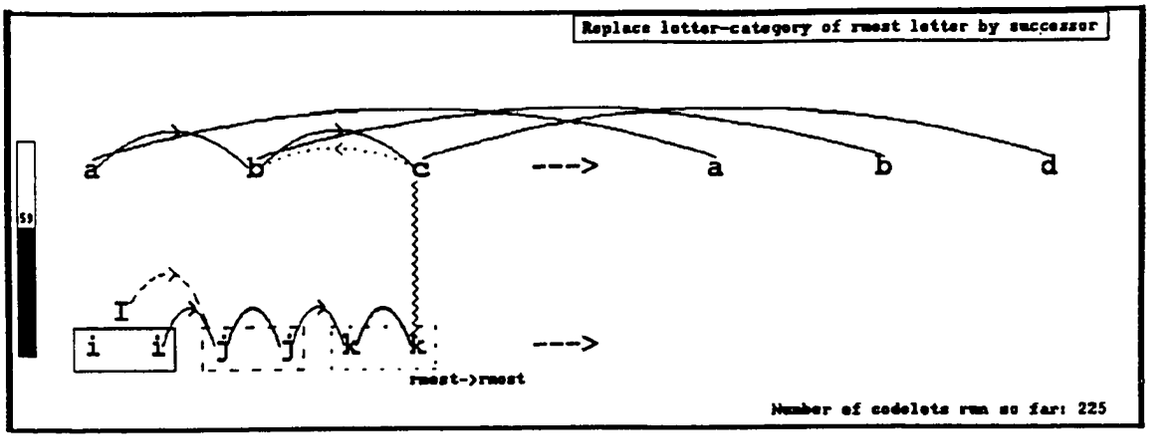
Slipnet: The activation of the nodes *successor* and *same* has caused top-down bond-scouts to be posted to look for more relationships of these types. These nodes have begun to spread activation to *successor-group* and *sameness-group* ("sm grp"), which will in turn post codelets to look for groups of these categories. The activation of the node *right* (the direction of the a–b successor bond) has caused top-down bond-scouts and top-down group-scouts to be posted to look for bonds and groups in this direction.



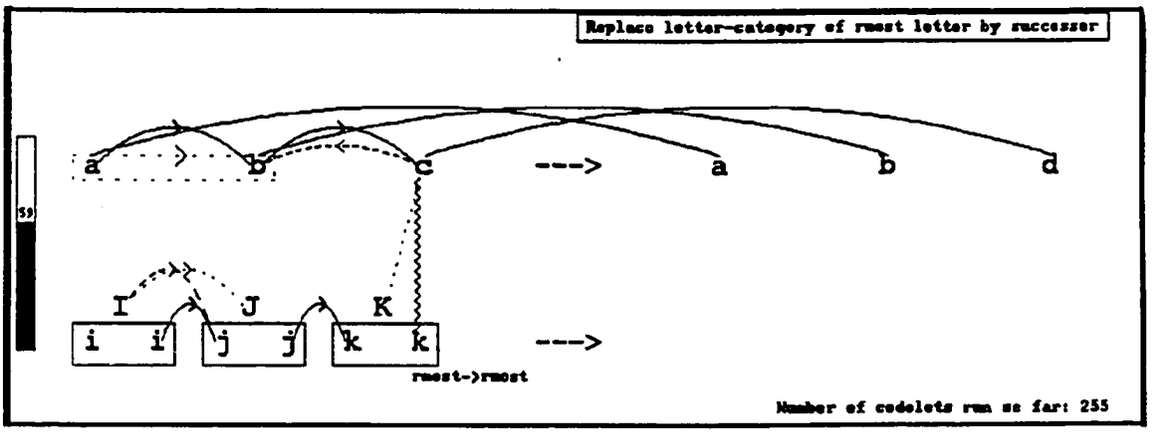
4. **Workspace:** After 165 codelets have run, some sameness groups are being considered (made up of the **i**'s and the **j**'s), and a correspondence between the **c** and the rightmost **k** has been built. Also, a rule, "Replace letter-category of rightmost letter by *D*", has been built. If the program were to stop right now, its answer would be **iijjkd** (which the program sometimes gets, as can be seen from the bar graph for this problem). This rule is relatively weak, though, and will soon face competition from a stronger rule.



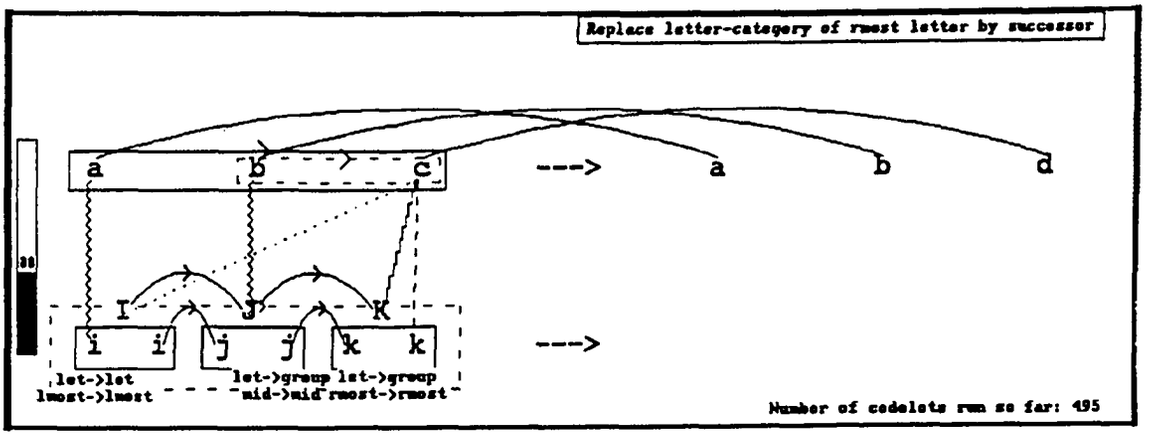
Slipnet: The nodes *successor* and *right* have decayed, but the nodes *sameness* and *sameness group*, being more of greater conceptual depth, remain highly active. For the time being, the program is concentrating more on finding and evaluating sameness bonds and sameness groups which, being intrinsically stronger, tend to be explored and built faster than other kinds of bonds and groups.



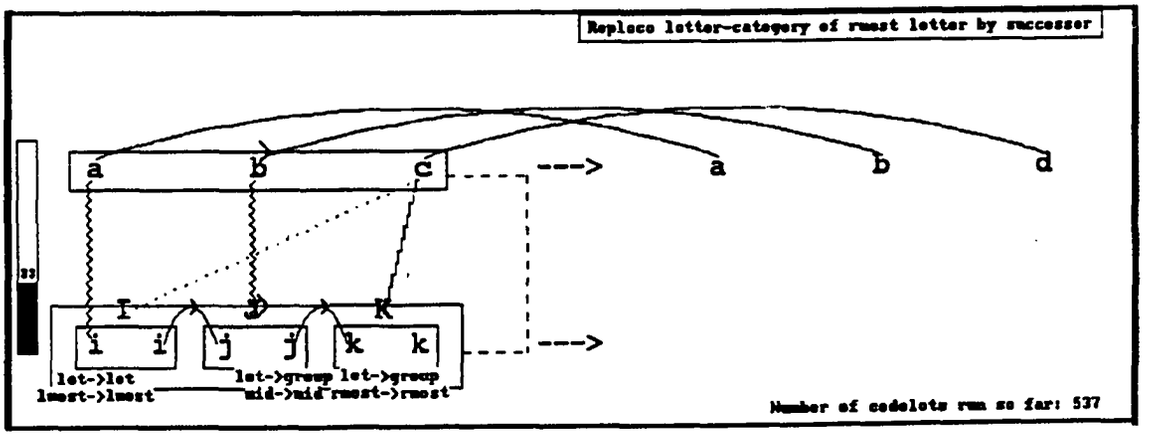
5. **Workspace:** 225 codelets into the run, successorship has taken hold as the fabric of the initial string. In the target string, a sameness group has been built out of the two i's, and two other such groups are being considered. A successor bond is being considered between the group **I** and its right neighbor, the letter **j** (perhaps due to top-down pressure from the activation of *successor* and *right*). The previous weak rule has been replaced by the stronger rule "Replace letter-category of rightmost letter by successor"; thus, if the program were to



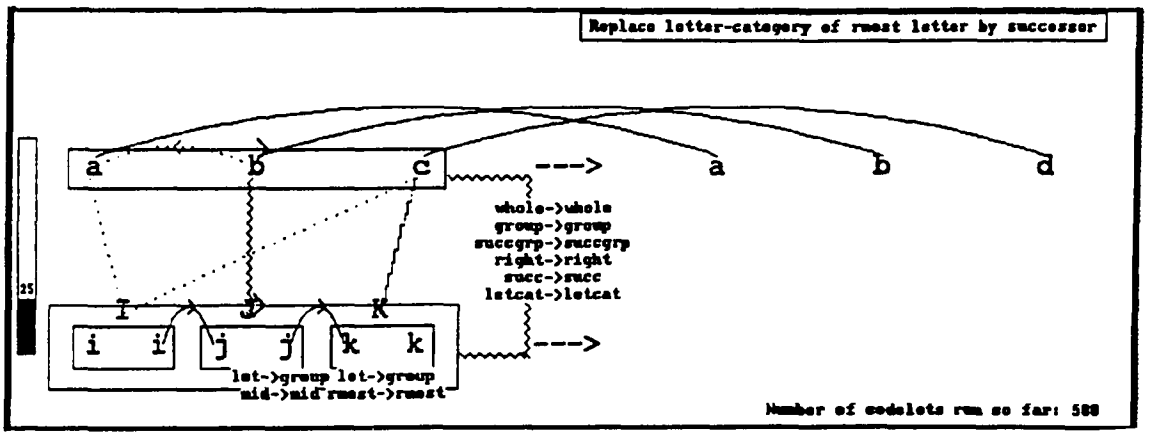
6. **Workspace:** All three sameness-groups have been built in the target string, and a successor bond is being considered between the group **I** and group **J** (which will compete with the proposed successor bond between the group **I** and the letter **j**). There is also a competition unfolding between the **c**–**k** correspondence and the **c**–**K** correspondence. The latter correspondence has a better chance, even though it involves a slippage, *letter* ⇒ *group*, whereas the former does not. There are two reasons for this: (1) Copycat has a bias towards correspondences involving larger objects (e.g., a group is larger than a single letter), and (2) the group **K** is now much more salient than either of its component letters, so there will be many more attempts to build the latter correspondence than the former, and statistics will tend to work in its favor.



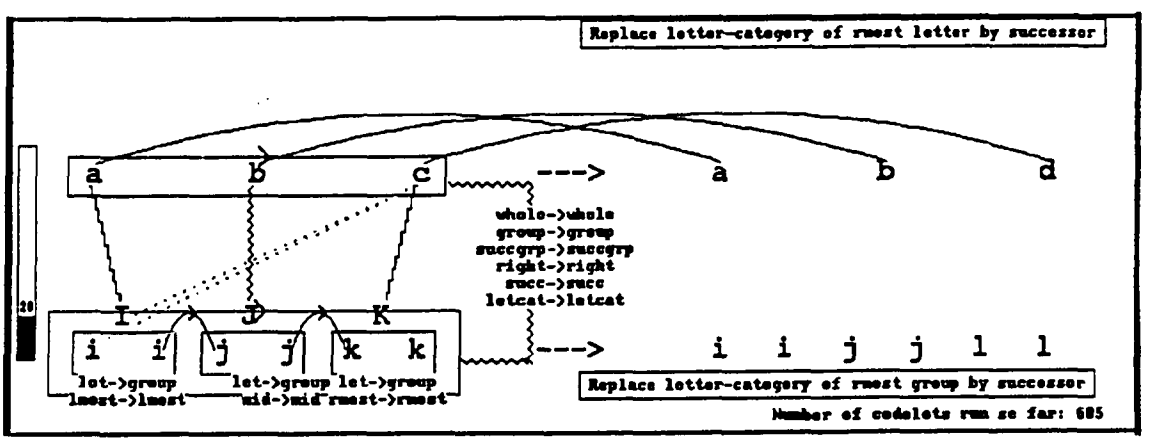
7. **Workspace:** Now, 495 codelets into the run, the initial string has been grouped as a right-going successor group, and strong top-down pressures from *successor* and *successor-group* have helped to accelerate a similar view of the target string, but at the level of groups rather than individual letters. The c–K correspondence has won over the c–k correspondence (though the latter is once again being considered). Also, a b–J correspondence has been built (notice that the program has at some point described the J group as "middle"). There is still a correspondence between the a and the leftmost letter i rather than the group I. The temperature has gone down to 38, reflecting the assessed promise of the structures that have been built so far. A "diagonal" c–I correspondence has been proposed, but it is very weak (it is based only on the weak concept-mapping *rightmost* ⇒ *leftmost*), and though a codelet for testing its strength has been posted, its urgency is very low, and is suppressed even further by the low temperature.



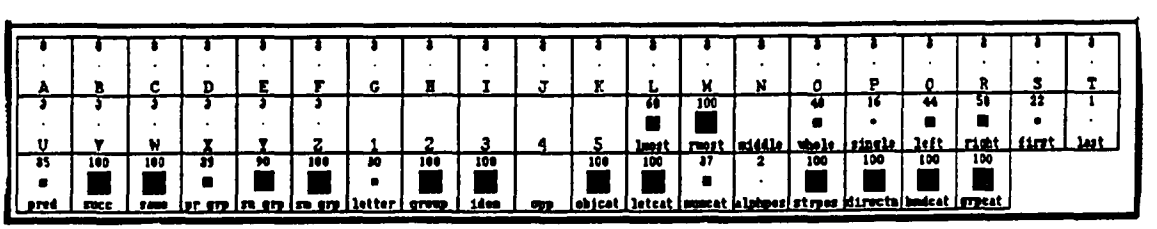
8. **Workspace:** The whole-target-string successor group has been built, and a correspondence is being considered between the two whole-string groups.



9. **Workspace:** The whole-string correspondence has been built. The **a–i** correspondence has been broken, and an **a–I** correspondence is being considered in its place. The temperature has fallen to 25.



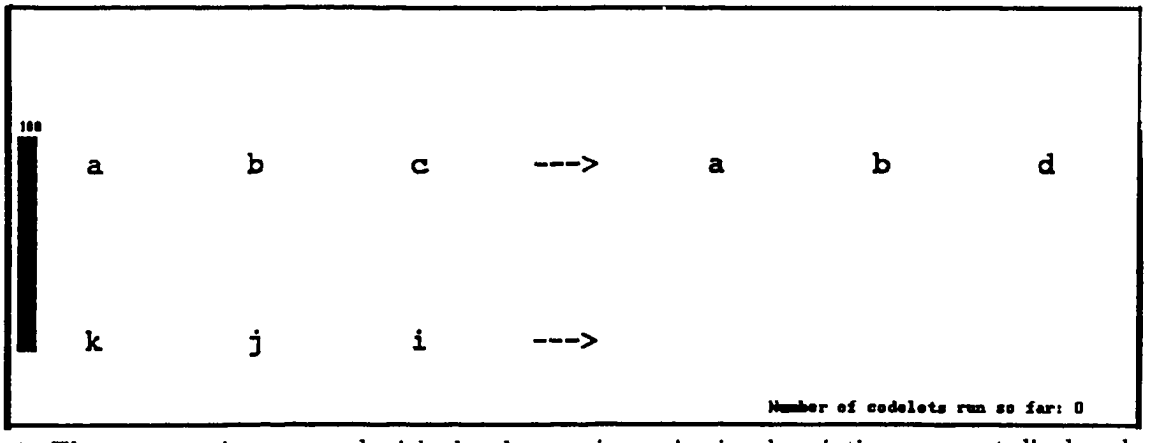
10. **Workspace:** The **a–I** correspondence has been built, the rule has been translated (according to the slippage *letter* ⇒ *group*), and the answer **iijjll** has been given. The low temperature of 20 reflects the program's satisfaction with this answer (though of course it isn't as low as it was for **ijk** ⇒ **ijl**, since there is a slippage here—namely, *letter* ⇒ *group*—that didn't have to be made in that problem).



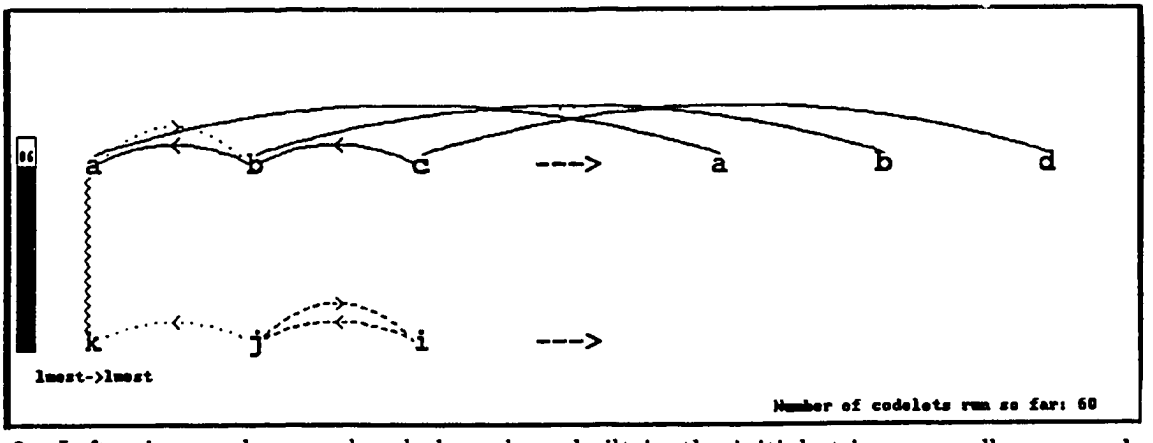
Slipnet: The final configuration of the Slipnet reflects what was important in this problem: not any particular letter category, but rather the notions of *rightmost*, *successorship*, *sameness*, *successor group*, *sameness group*, and so on.

abc ⇒ abd, kji ⇒ ?—Run 1

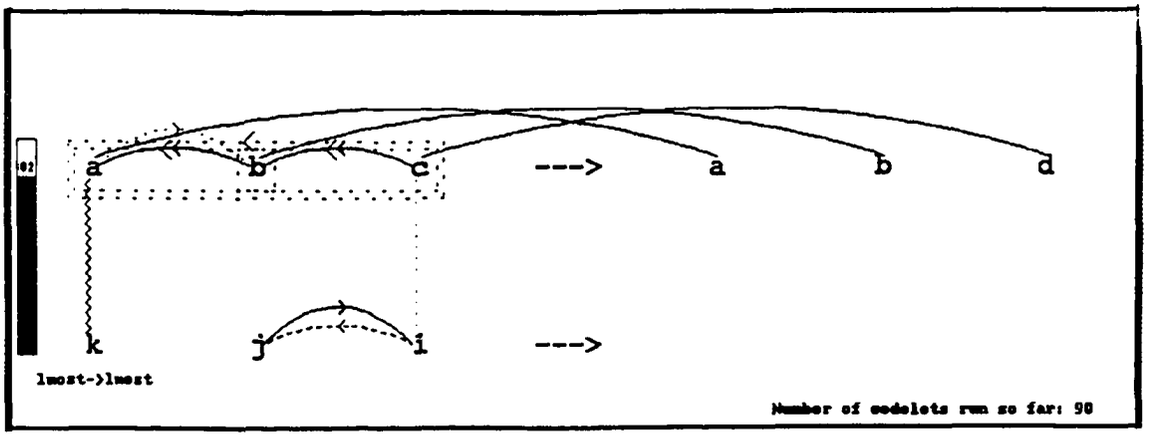
The following is a set of screen dumps from a fairly typical run of Copycat on this problem. The Slipnet is not shown on this run.



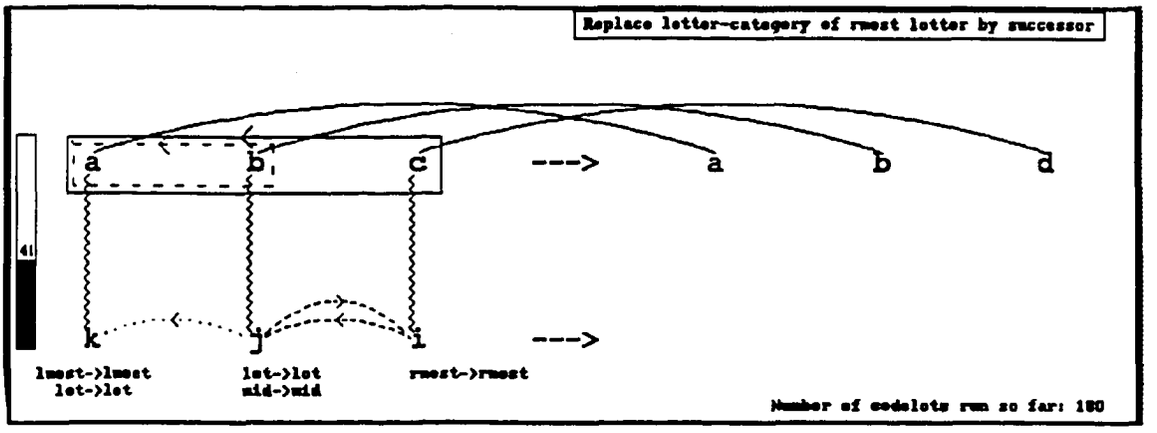
1. The program is presented with the three strings. Again, descriptions are not displayed; they are the same as in the problem "**abc ⇒ abd, ijk ⇒ ?**".



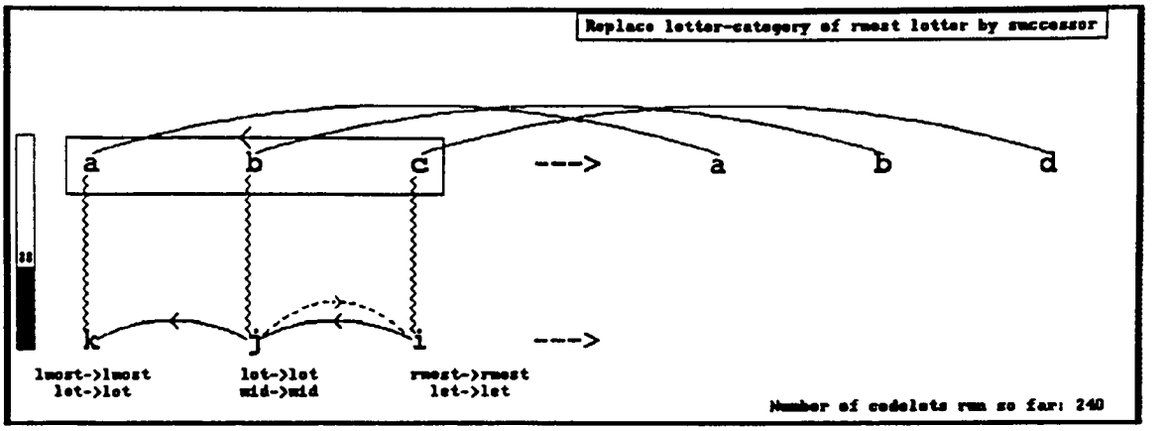
2. Left-going predecessor bonds have been built in the initial string, as well as an a–k correspondence, causing the temperature to fall to 86. The resulting activation of *predecessor* and *left* in the Slipnet creates pressures for the program to see predecessor bonds and left-going bonds, but, unlike in the previous two problems, these two pressures cannot be satisfied simultaneously: the initial and target strings run in different alphabetic directions. There is thus competition between these pressures in the target string, with the pressure to see predecessor bonds being stronger than the pressure to see left-going bonds, since the former is has greater conceptual depth. But there is another set of very strong pressures that rivals this: *leftmost* ⇒ *leftmost* and *rightmost* ⇒ *rightmost* correspondences attempt to enforce a view in which bonds in the two strings have the same direction, since they are incompatible with the *left* ⇒ *right* and *right* ⇒ *left* slippages that would result from a view in which bonds in the two strings were in opposite directions.



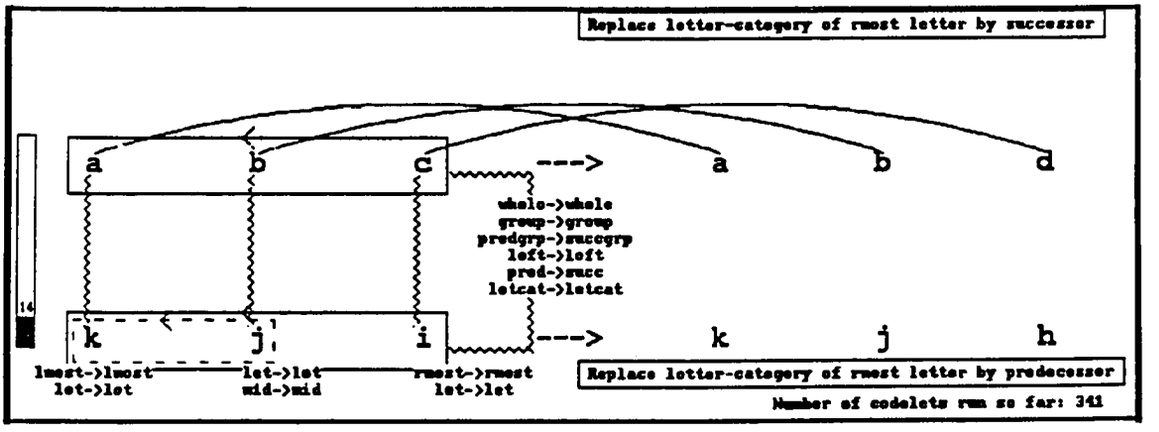
3. Some groups are being considered in the initial string, the strongest of which is the whole-string predecessor group. A j–i predecessor bond has been built in the target string, in response to top-down pressure from *predecessor*. But the proposed vertical c–i correspondence will fight against it, since the concept-mapping *rightmost* ⇒ *rightmost* is incompatible with the existence of bonds on the two sides of the correspondence going in opposite directions.



4. A whole-string left-going predecessor group has been built in the initial string. The *rightmost* ⇒ *rightmost* correspondence has won, breaking the j–i bond. This "vertical" (i.e, *leftmost* ⇒ *leftmost*, *rightmost* ⇒ *rightmost*) correspondence viewpoint is working hard to force the program to build bonds all in the same direction, in spite of the strong pressure from *predecessor*, which remains active in the Slipnet, and which lobbies for building predecessor bonds going in opposite spatial directions.



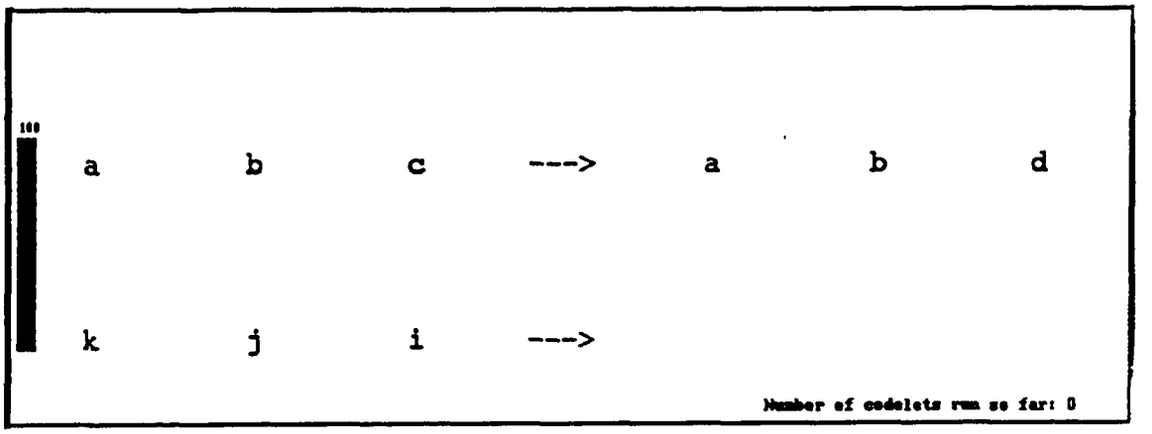
5. The same-direction pressure is prevailing, with the strong set of vertical correspondences remaining intact, and left-going successor bonds being built in the target string. The temperature is already fairly low, making it unlikely that this viewpoint will be destroyed at this point, even though there are still attempts being made to build predecessor bonds in the target string.



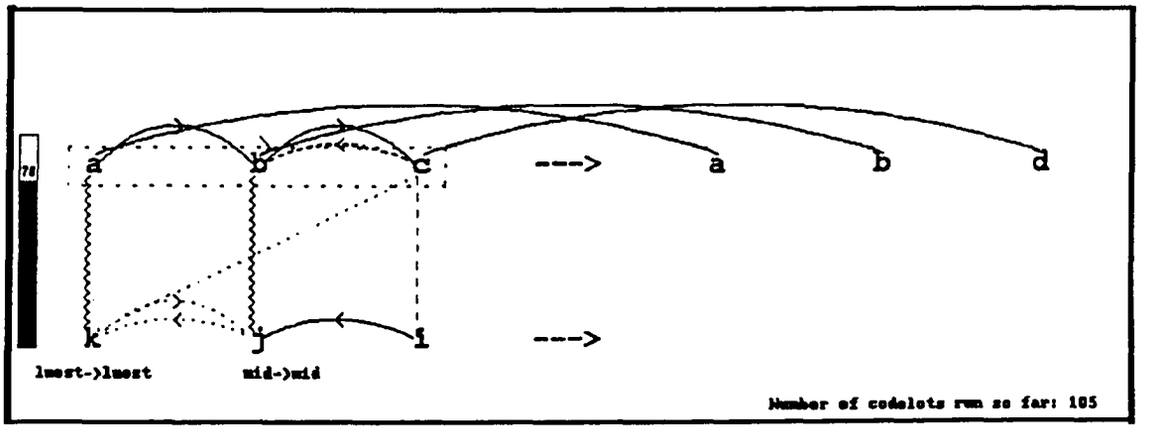
6. A left-going successor-group has been built in the target string, a correspondence has been built between the two whole-string groups, involving the slippages *predecessor-group* ⇒ *successor-group* and *predecessor* ⇒ *successor*. The rule has been translated according to those slippages ("Replace letter-category of rightmost letter by *predecessor*"), and the answer **kjh** has been given.

abc ⇒ abd, kji ⇒ ?—Run 2

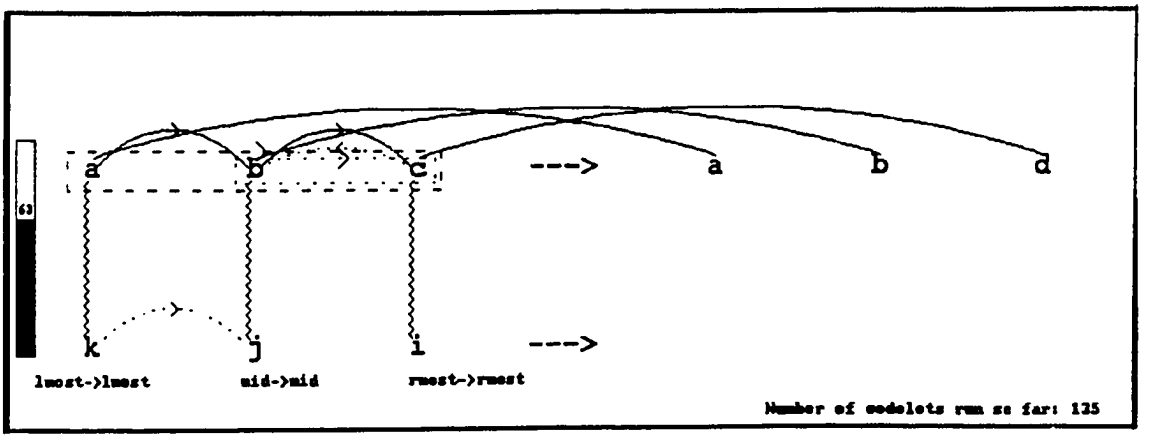
The following is a set of screen dumps from different run of Copycat on the same problem, leading to a different answer.



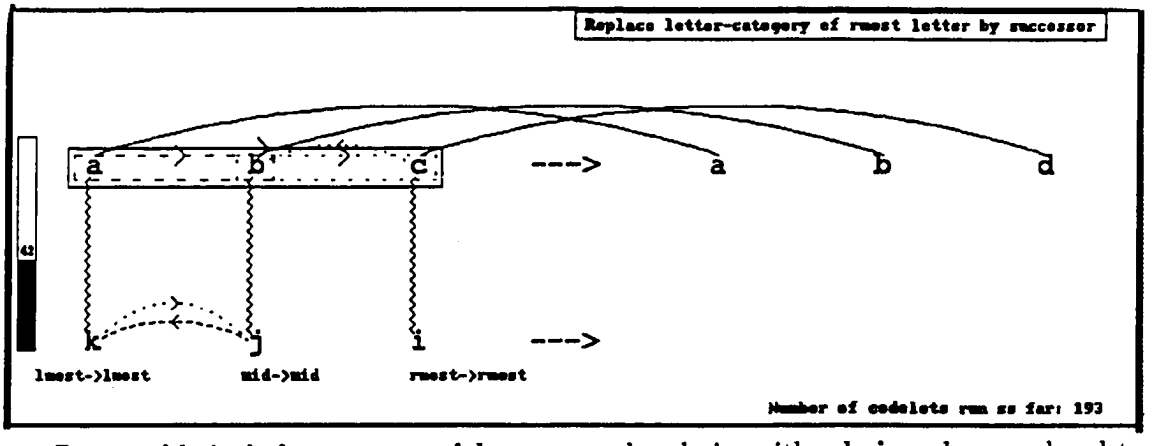
1. The program is presented with the three strings.



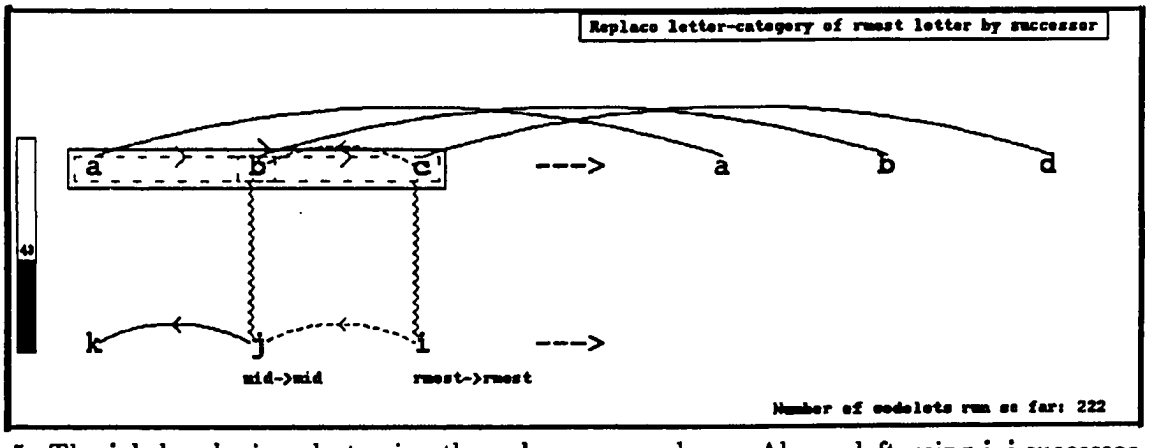
2. We skip ahead 105 codelets into the run. Right-going successor bonds have been built in the initial string. As in the previous run, this sets up two opposing top-down pressures for the target string: a pressure to see successor bonds (which in the target string are *left-going*), and a pressure to see right-going bonds (which in the target string are *predecessor* bonds). Various proposed bonds are being considered, and a left-going **i–j** successor bond has been built. As in the previous run, the vertical correspondences lobby for building target-string bonds in the same direction as those of the initial string. Some fights are in store—in particular, between the strong proposed **c–i** correspondence (which is supported by the already-built **a–k** correspondence) and the **i–j** bond (which is supported by the activation of the node *successor*).



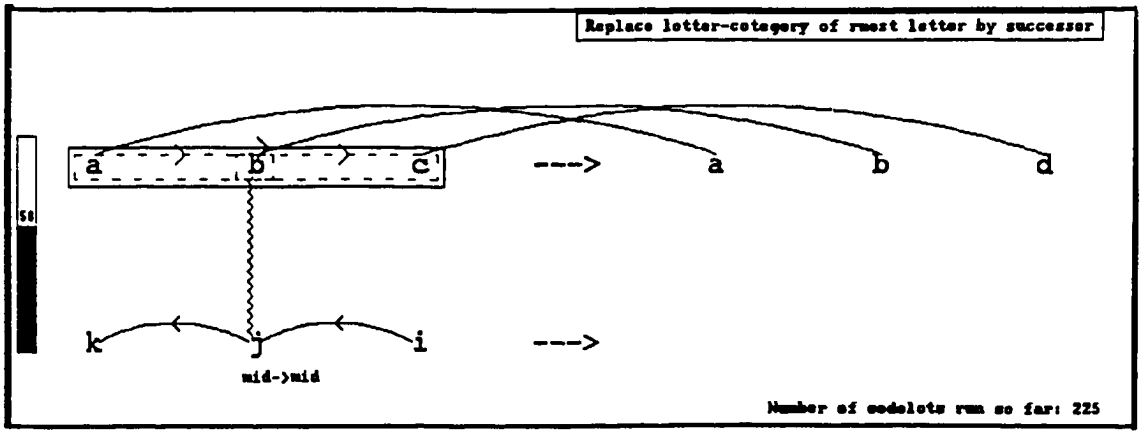
3. The c–i correspondence has been built, destroying the i–j bond in the process. At this point it looks like the same-direction view is going to win out, as it did in the previous run.



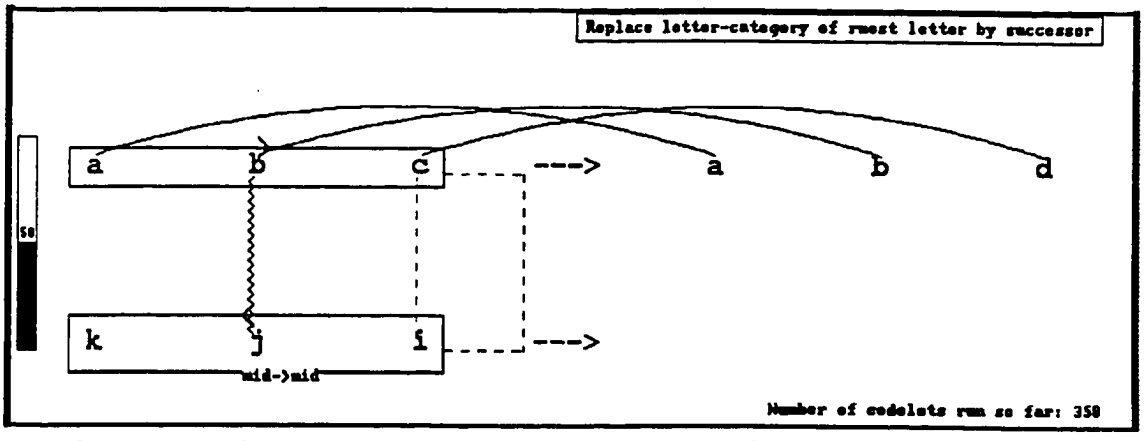
4. But trouble isn't far away, as a j–k successor bond vies with a k–j predecessor bond to be built.



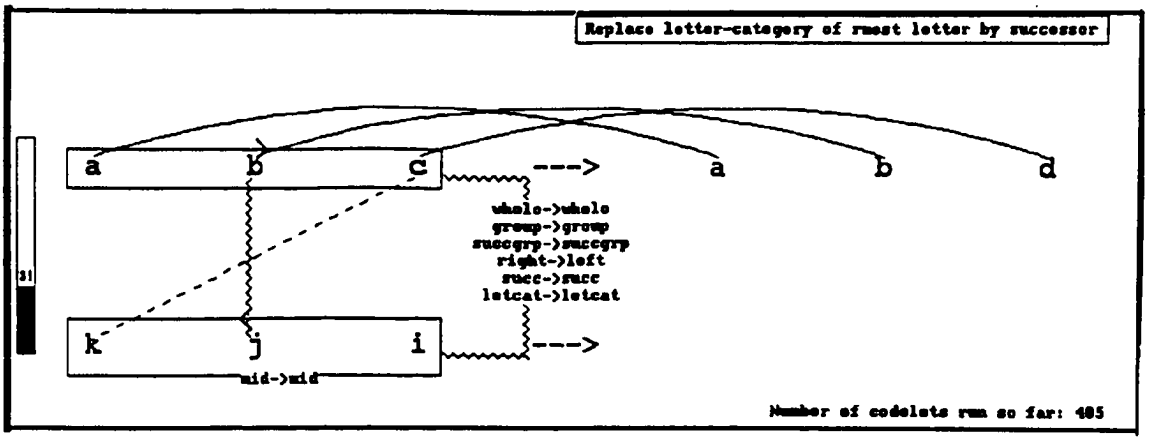
5. The j–k bond wins, destroying the a–k correspondence. Also, a left-going i–j successor bond is now waiting to be built as well.



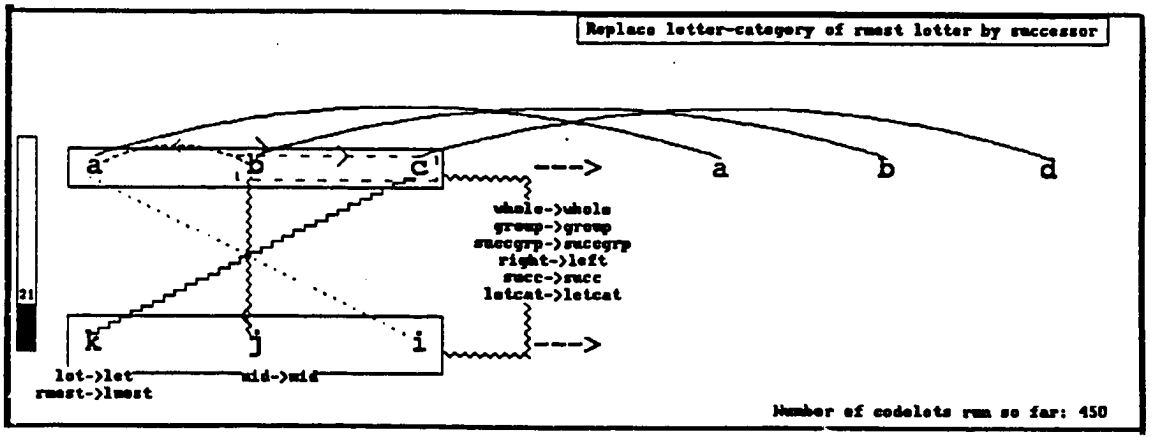
6. The i–j bond has been built, also destroying a correspondence, and now it looks like the successorship viewpoint is going to win the day. The temperature has gone up to 58 because a correspondence was broken.



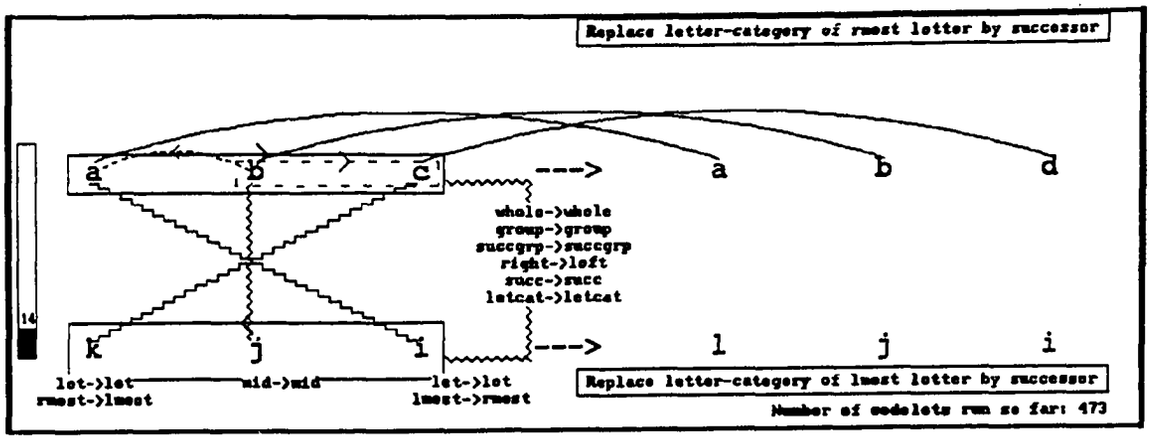
7. The successorship viewpoint is becoming more entrenched, for the following reasons: the target string has been perceived as a successor group; a correspondence is being considered between the two groups as wholes; and the temperature is falling (it is now 50). Even so, a c–i correspondence (which would contradict the opposite-direction whole-group correspondence) is being considered.



8. A correspondence is built between the two successor-groups, involving a *right* ⇒ *left* slippage. This activates *opposite*, and the combination of that slippage and *opposite*'s activation gives a great deal of support to a proposed diagonal c–k correspondence, with slippage *rightmost* ⇒ *leftmost*.



9. The c–k correspondence has been built, and this (along with the continuing activation of *opposite*) creates further support for an a– i, *leftmost* ⇒ *rightmost* correspondence. A **bc** group has been proposed, challenging the much stronger **abc** group, but given the low temperature, it is very unlikely that it will get anywhere at this point.



10. A strong, mutually supporting set of correspondences has been built, resulting in a low temperature of 14. The rule has been translated according to the instructions in the concept-mappings, as "Replace letter-category of *leftmost* letter by successor", yielding answer **lji**.

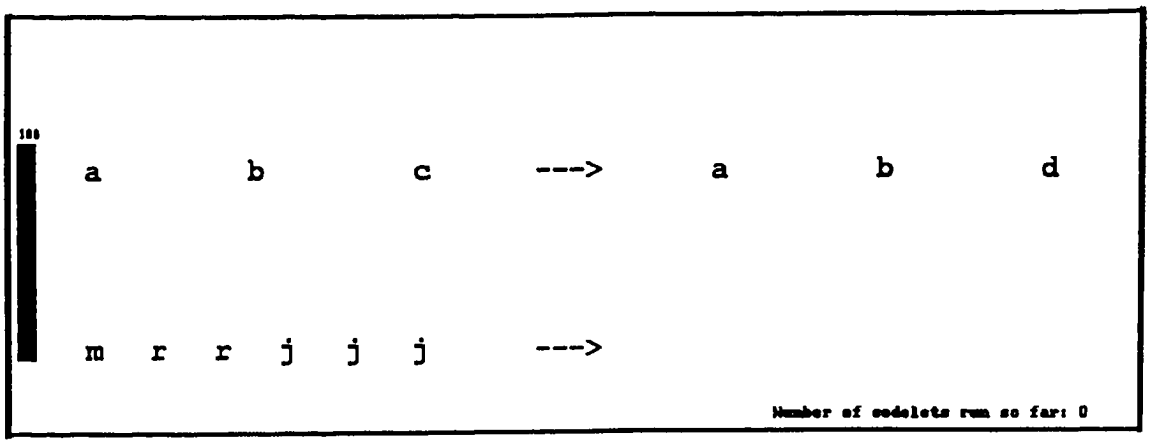
abc ⇒ abd, mrrjjj ⇒ ?

The set of screen dumps given below shows one way in which Copycat arrives at the answer **mrrjjjj** (and thus they are from a not-so-typical run on this problem; this answer is given about 4% of the time). This problem is different from the problem "**abc ⇒ abd, xyz ⇒ ?**" (shown in the next section) in that there is no obvious "snag" (such as the fact that *Z* has no successor) blocking a good answer. Rather, the straightforward answer **mrrkkk** just doesn't seem very strong, since there are no bonds tying together the target string as a whole. In particular, the strong and seemingly explanatory successorship structure in **abc** is completely lacking in **mrrjjj** when only the letter-categories are considered. Copycat usually simply gives up and produces one of the two more obvious answers, even though the temperature remains fairly high (most people also give one of these two answers). But on some more interesting runs (such as the one shown here), it does manage to see the relations between the lengths of the groups in the target string, and to produce **mrrjjjj**.

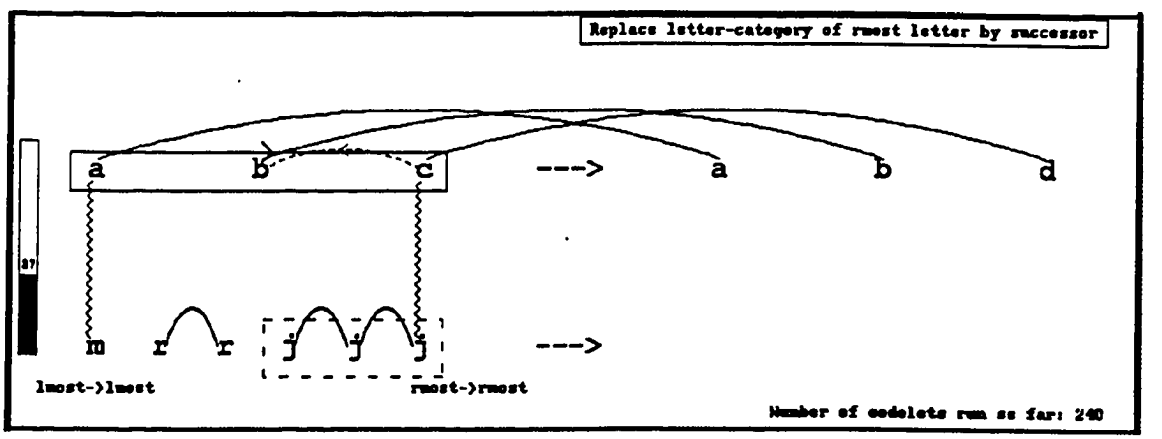
The point here is to illustrate how a number of pressures interact to allow the notion of *group length*, which in most problems remains essentially dormant, to come to be seen as relevant in this problem. On most runs, the groups **rr** and **jjj** are constructed. As happened in **iijjkk**, each group's letter-category (*R* and *J* respectively) is explicitly noted, since letter-category is relevant by default. By contrast, although there is *some* probability that lengths will be noticed at the time the groups are made, it is low, since *length* is not normally strongly associated with the concept of *group*. Once the groups **rr** and **jjj** are

made, the concept *sameness-group* becomes very relevant. This creates top-down pressure for the system to describe other objects—especially in the same string—as sameness groups if possible. The only way to do this here is to describe the single **m** as a "sameness group" with just one letter. This is strongly resisted by an opposing pressure: a single-letter group is an intrinsically weak and farfetched construct. It would be disastrous for the program if it were willing to bring in unlikely notions such as single-letter groups without strong pressure: the program would then waste huge amounts of time exploring unlikely possibilities in *every* problem. As was discussed in Chapter 2, given the limitation of time and cognitive resources one has in real life, it is absolutely vital to resist looking at situations in nonstandard ways unless there is strong pressure to do so.

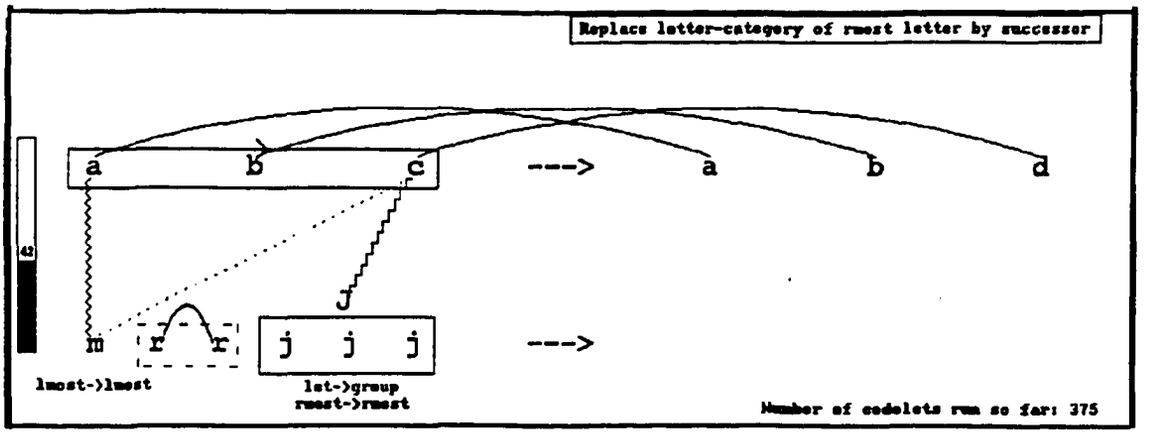
Copycat resists farfetched notions such as single-letter groups, but in this problem, the existence of two other groups in the string, coupled with the lone **m**'s unhappiness at its failure to be incorporated into any large, coherent structure, pushes against this resistance. These opposing pressures fight; the outcome is decided probabilistically. If the **m** winds up being perceived as a single-letter group, its length will very likely be noticed (single-letter groups are noteworthy precisely because of their abnormal length), making *length* more relevant in general, and thus increasing the probability of noticing the other two groups' lengths. Moreover, *length*, once brought into the picture, has a good chance of staying relevant, since descriptions based on it turn out to be useful. (Note that had the target string been **mrrrrjj**, length might be brought in, but it would not turn out useful, so it would likely fade back into obscurity.) In **mrrjjj**, once lengths are noticed, the successor bonds among them can then be constructed by bond scouts that are continually seeking new bonds—in particular, by top-down successor scouts resulting from the already-seen successor bonds in **abc**. Thus the crux of discovering this solution lies in the triggering of the concept of *length*.



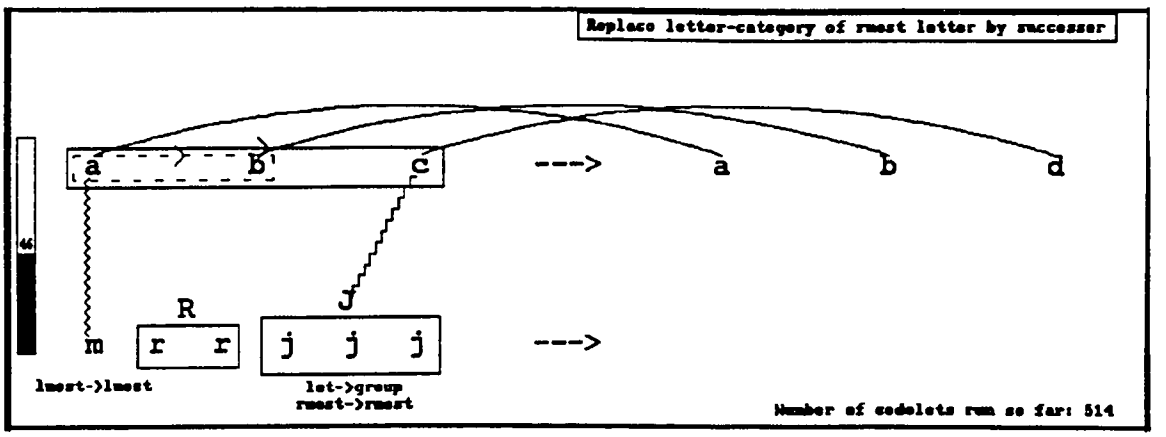
1. The program is presented with the three strings. (The Slipnet will not be displayed on this run, though aspects of its state will be described from time to time.)



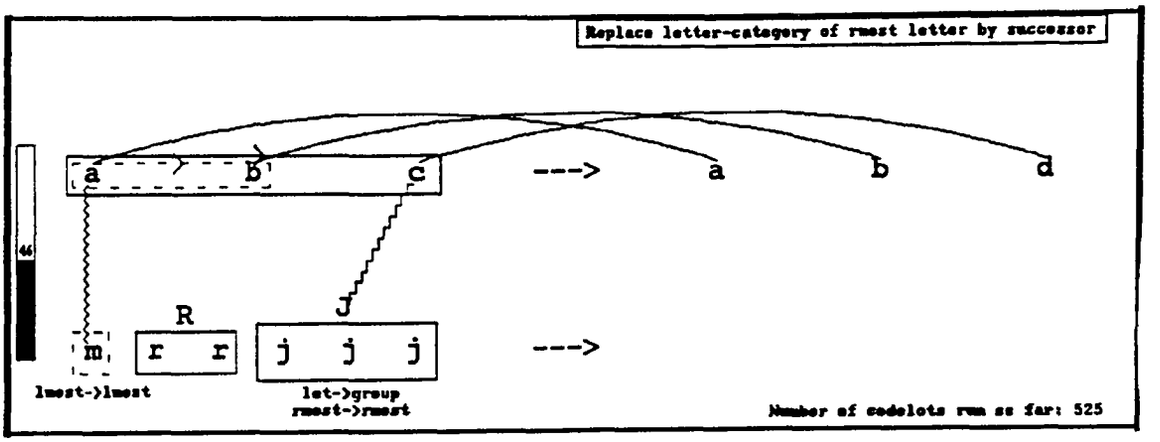
2. We skip ahead to 240 codelets into the run. Much progress has been made: a whole-string successor group has been built in the initial string, sameness bonds have been built in the target string, a **jjj** sameness group is being considered, correspondences have been built between the two leftmost and two rightmost letters in each string, and a rule has been built. The temperature has fallen to 37. If the program were to stop at this point, the answer would be **mrrjjk**.



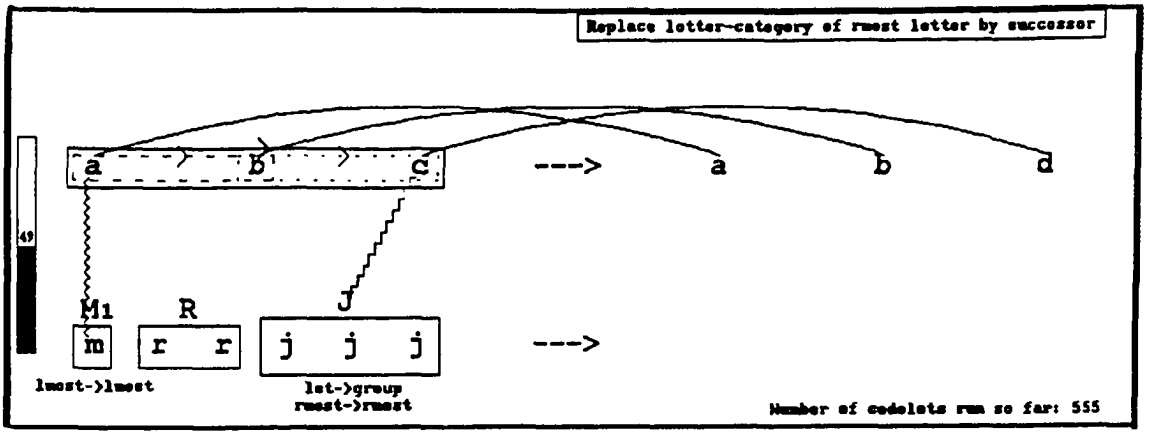
3. The group **J** has been built, creating more top-down pressure to see sameness groups in the string, and a correspondence has been made between the **c** and it. (If the program were to stop at this point, the answer would be **mrrkkk**.) A grouping of the two **r**'s is being considered, as is a weak diagonal correspondence between the **c** and the **m**. The temperature has gone up a bit for two reasons. First, the **c**–**J** correspondence is not as strong as the previous **c**–**j** correspondence because the **a**–**m** correspondence more strongly supported the latter. Second, the creation of a new object (here, the group **J**) can cause the temperature to go up, since temperature is a function of the happiness of all the objects, and while the existence of the group increases the happiness of its members, it itself starts out unhappy (e.g., the group **J** has no bonds to anything else in its string). The result can add up to an increase in total unhappiness. This initial unhappiness is a necessary thing: it serves to quickly attract codelets to the new object.



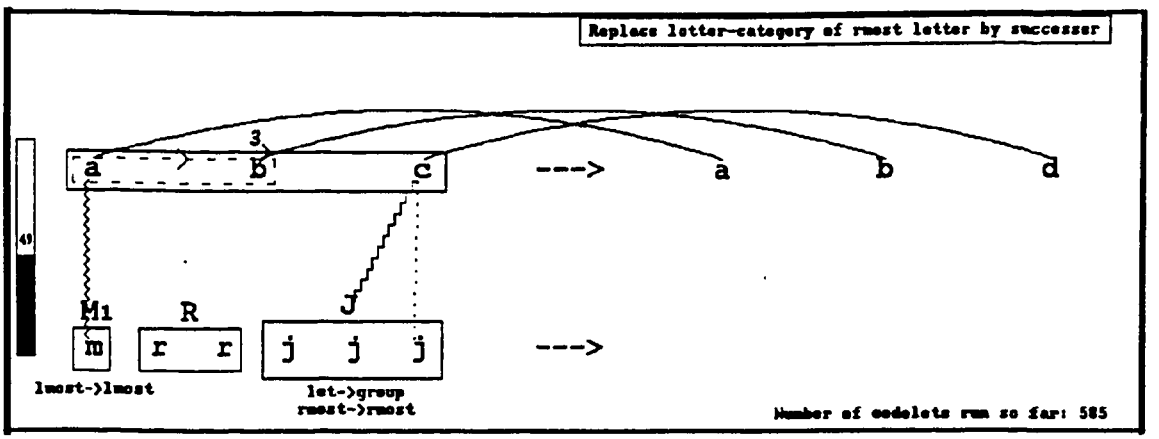
4. The group of r's has been built. There are several pressures at work at this point. First, the single **m** remains unhappy since it is not integrated into any structures in its string. Second, the groups **R** and **J** remain unhappy, because in spite of many tries by various bond scouts, especially top-down bond scouts trying to make successor bonds, no bonds can be made between them. This continuing unhappiness keeps the temperature relatively high. Finally, the presence of two sameness groups in the target string, as well as the high activation of the node *sameness-group*, creates strong pressure to see more such groups in the target string.



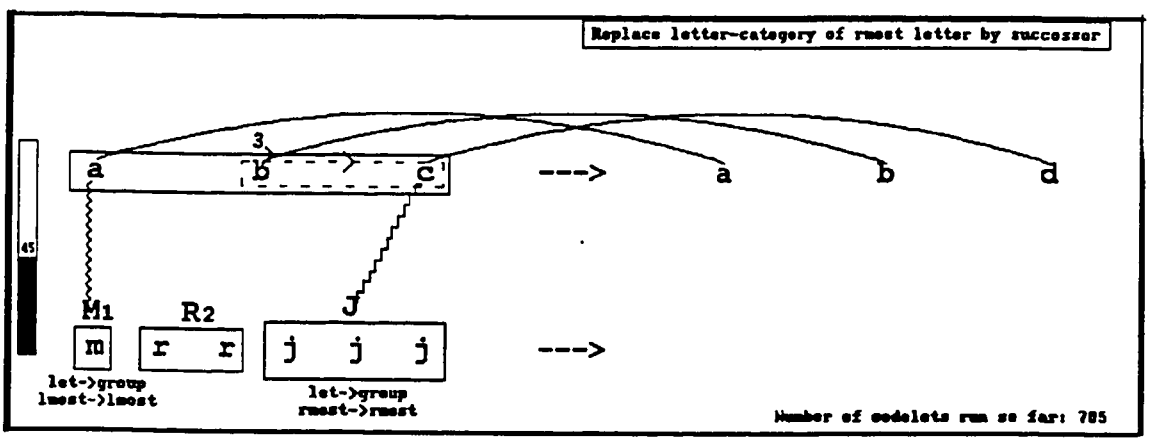
5. The strong pressures described in the last caption have overcome the intrinsic resistance to proposing a sameness group consisting of a single letter, and such a group, consisting of the single **m**, has indeed been proposed (dashed rectangle around the **m**). Top-down group-category scouts can propose such groups, but such a proposal is intrinsically very unlikely and almost never happens unless there are strong pressures that make it more likely. The probability of proposing a single-letter group is a function both of the amount of local support in the string and of the activation of *number-category*—i.e., if group lengths have already been deemed to be important, then it is more likely that single-letter groups can be proposed. Here, with group lengths not yet in the picture, the proposal of such an oddball group is a result of a combination of factors: unhappiness of the lonely single letter (which makes it salient, causing lots of codelets to concentrate on it, so after many tries, one may succeed) along with lots of local support for such a group in the string (the principle of "safety in numbers" discussed in Chapter 3) and relatively high temperature (making intrinsically unlikely events a bit more likely).



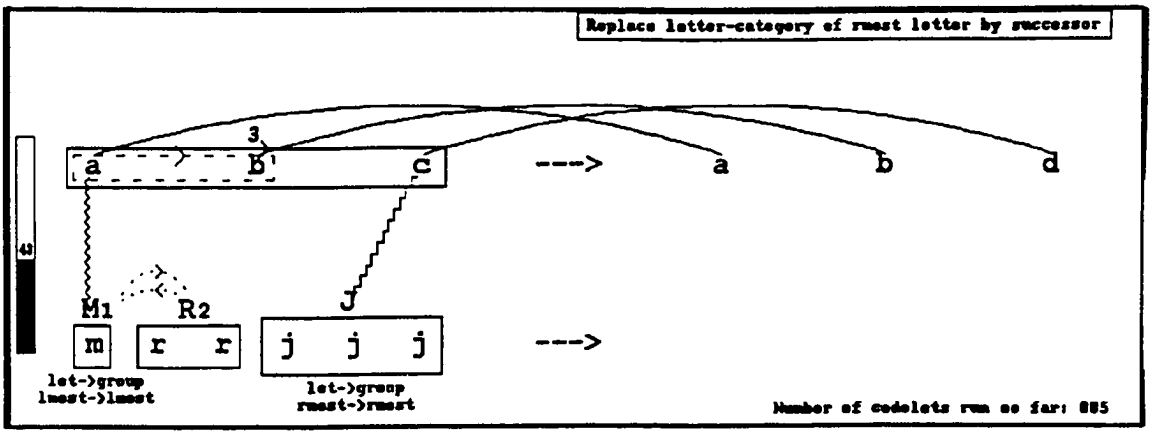
6. The single-letter group has been built, and its length has been noticed (displayed as a "1" next to the M). There are two ways in which group lengths can be noticed in Copycat. The first way is for a group-builder codelet to attach a length description at the time the group is built. A group-builder codelet always has some probability of doing this, the probability being a function of both the length of the group (the shorter, the more probable, with probability dropping off very quickly with increasing length) and the activation of *number-category* (when it is relevant, noticing length is much more likely). So *a priori*, there is not much likelihood for a group-builder to notice the lengths of two-element groups, less for three-element groups, and so on. But it is rather likely that a group-builder will notice the length of single-letter groups, since it is precisely their short length that makes them noteworthy. Length descriptions can also be attached to already-formed groups (e.g., the **rr** group here) by top-down description-scout codelets, posted by *number-category* once it becomes activated, as it is now, as a result of spreading activation from the node *1*, which was activated when the single-letter group was formed and its length was noticed. (The probability of creating a length description in either of these two ways is of course also dependent on temperature.) The activation of *number-category* means that *length* is now a relevant notion, which creates pressure on the program (in the form of top-down description-scouts) to continue to use *length* as an organizing theme. If *length* does not turn out to be a useful notion, *number-category*'s activation decays, and this pressure subsides.



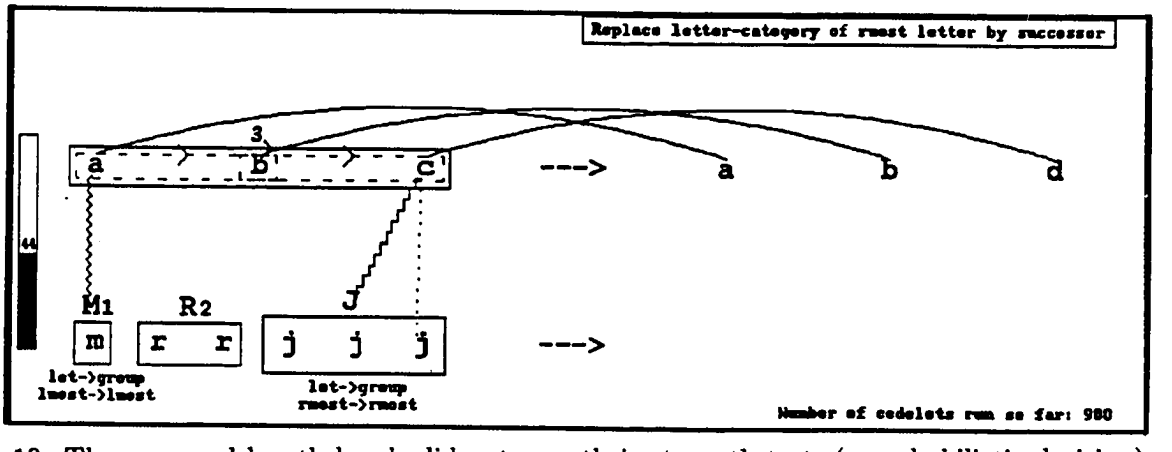
7. A length description of *3* has been attached to the **abc** group by a top-down description scout posted by *number-category*, which remains active.



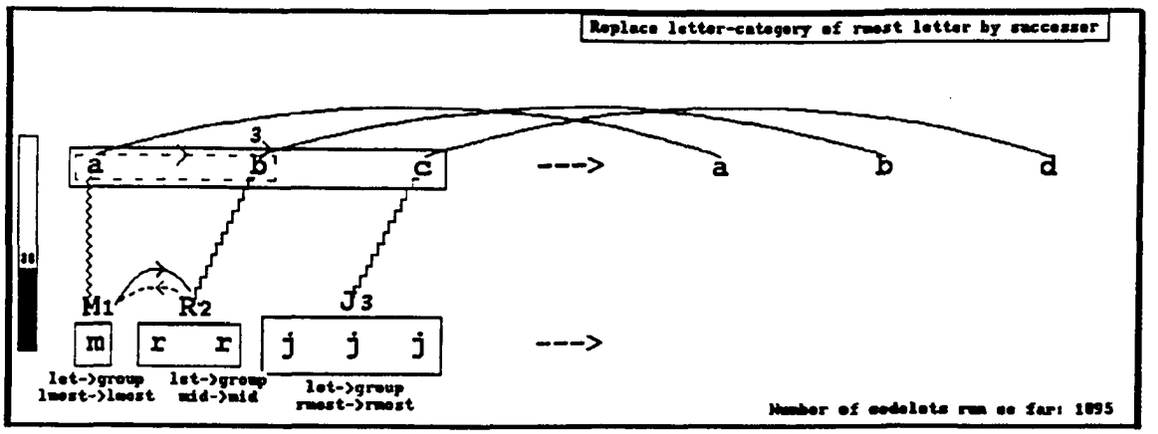
8. The activation of *number-category* and the existence of the description *1* created pressure for length descriptions in the target string (the existence of the *1* makes it more likely that description-scout codelets will succeed in building other such descriptions—again, the principle of safety in numbers). As a result, a length description has been attached to the group of **r**'s. Also a correspondence has now been built from the **a** to the group **M** (a subtle change from the previous **a**-to-letter-**m** correspondence).



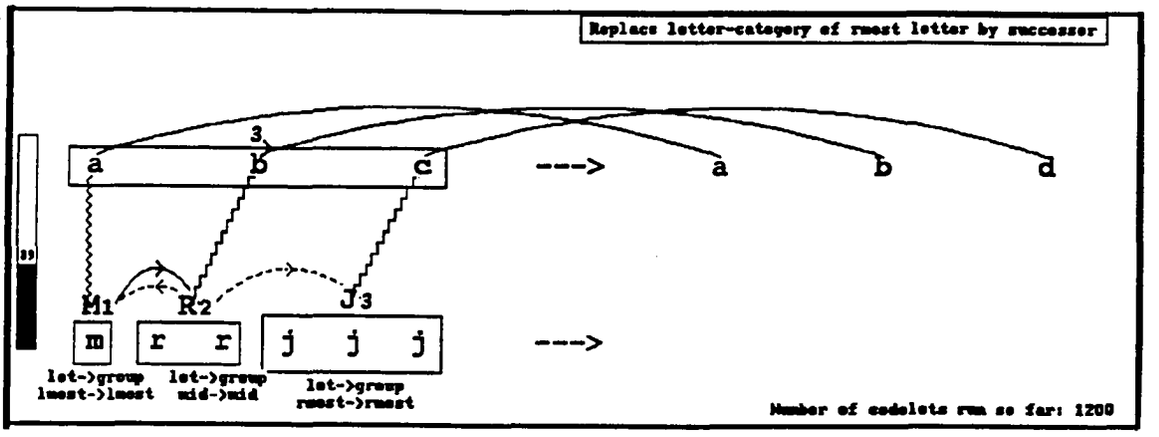
9. Some proposed bonds are being considered between the lengths of the M and R groups. There is some resistance to building these bonds—being less standard, bonds between lengths are not as strong as bonds between letter-categories (an *a priori* bias given to the program). Safety in numbers is again a principle here, and the lack of other length bonds in the target string increases the resistance to them.



10. The proposed length bonds did not pass their strength tests (a probabilistic decision) and have fizzled.

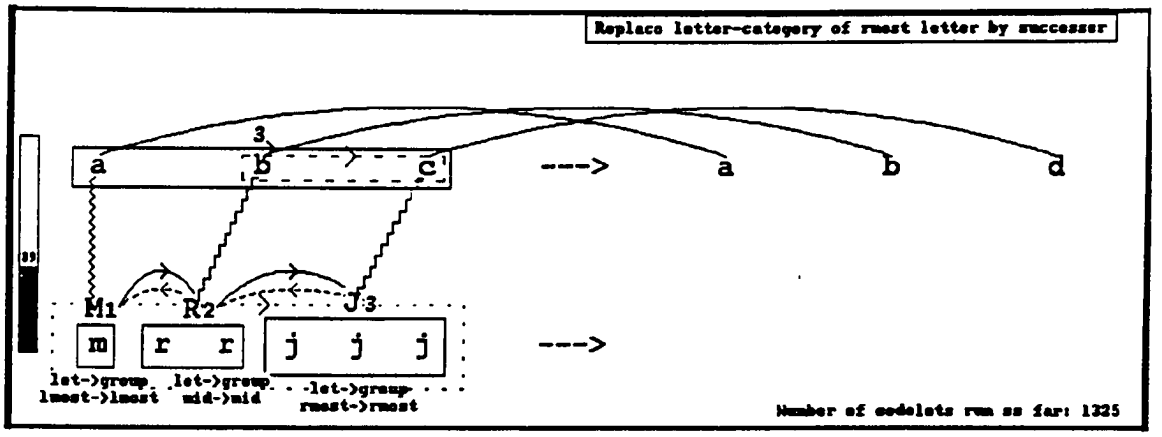


11. Try, try again. This time, the relatively high temperature, the top-down pressure from *successor*, and some persistence on the part of the program (note the time lapse of 390 codelets between the time when lengths were first noticed and now) have combined for success: a successor bond has finally been built between the group **M** and the group **J** on the basis of length. In addition, the group **J** has now been given the length description *3*. This also came about as a consequence of top-down pressure, safety in numbers, and persistence.

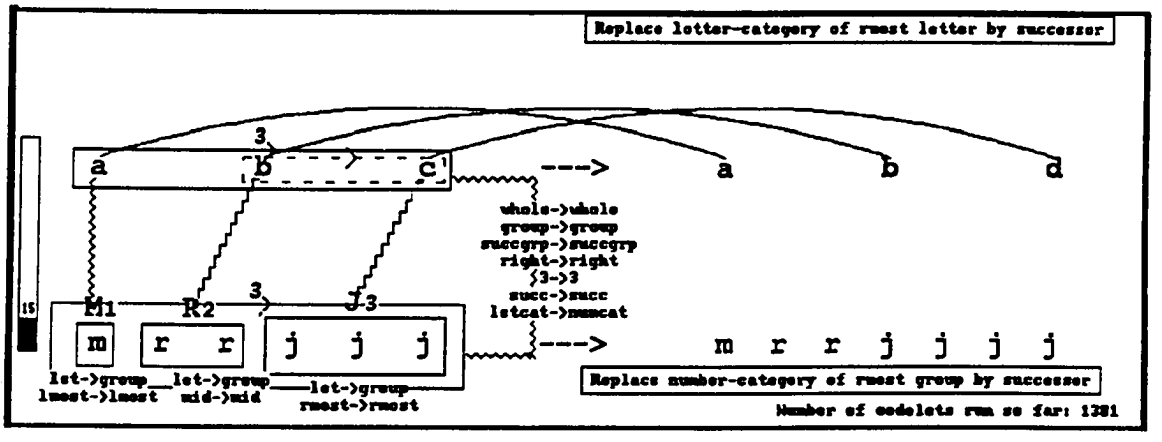


12. Top-down pressure at work again resulted in a *2–3* bond now waiting to be built. This time, building the length bond will be much easier, since another one already exists in the same string, giving local support to the new proposed bond.

Notice that over 100 codelets have run since the previous screen dump: the temperature is still relatively high and the program is still exploring a number of different possibilities (e.g., trying to build successor bonds among the letter-categories of the target-string groups, or to use the notion of *alphabetic-position*), none of which are panning out.



13. The second length bond has now been built in the target string, and a grouping of the whole target string, based on the successorship bonds, is being considered. In the Slipnet (not displayed here), the activation of the nodes *sameness* and *sameness-group* have faded, since these concepts are no longer very relevant to what is going on; instead, *successor*, *successor-group*, and *number-category* have taken over as the main organizing themes.



14. The whole-string group has been built in the target string and a correspondence has been made between the two strings as wholes, with the slippage *letter-category* ⇒ *number-category* (the respective description types that the groups' bonds were based on). The temperature has fallen to the low value of 15, indicating the program's satisfaction with this way of structuring the problem. The rule has been translated according to the slippages *letter-category* ⇒ *number-category* and *letter* ⇒ *group* to yield "Replace *number-category* of rightmost *group* by successor", yielding the answer **mrrjiii**.

Although this run may have looked quite smooth, there were many struggles involved in coming up with this answer: it was hard not only to make a single-letter group, but also to bring the notion of *group-length* into the picture, and to build bonds between group lengths. The program, like people, usually gives up before all these hurdles can be overcome, and gives one of the more obvious answers. Arriving at the deeper answer **mrrjjjj** requires not only the insights brought about by the strong pressures in the problem, but also a large degree of patience and persistence in the face of uncertainty.

The moral of all this is that in a complex world (even one with the limited complexity of Copycat's microworld), one never knows in advance what concepts may turn out relevant in a given situation. It is thus imperative not only to avoid dogmatically open-minded search strategies, which entertain all possibilities equally seriously, but also to avoid dogmatically closed-minded search strategies, which in an ironclad way rule out certain possibilities *a priori*. Copycat opts for a middle way, which of course leaves open the potential for disaster (as can be seen in the occasional bizarre answers it gets). This is the price that must be paid for flexibility. People, too, occasionally explore and even favor peculiar routes. The program, like people, has to have the potential to concoct crazy and farfetched solutions in order to be able to discover subtle and elegant ones like **mrrjjjj**. To rigidly close off any routes *a priori* would necessarily remove critical aspects of Copycat's flexibility. On the other hand, the fact that Copycat so rarely produces really farfetched answers demonstrates that its mechanisms manage to strike a pretty effective balance between open-mindedness and closed-mindedness, imbuing it with both flexibility and robustness.

These screen dumps show one way in which Copycat can arrive at **mrrjjjj**, but there are other ways as well. For example, it could first notice the relationship between the lengths of the **R** and **J** groups, which would then create very strong pressure for creating a single-letter group. Part of Copycat's flexibility rests in the fact that there are a number of different ways in which which it can arrive at each of the different answers to any problem. Not only are there a huge number of *microscopic* pathways to a given answer, but there are also a number of *macroscopic* pathways as well.

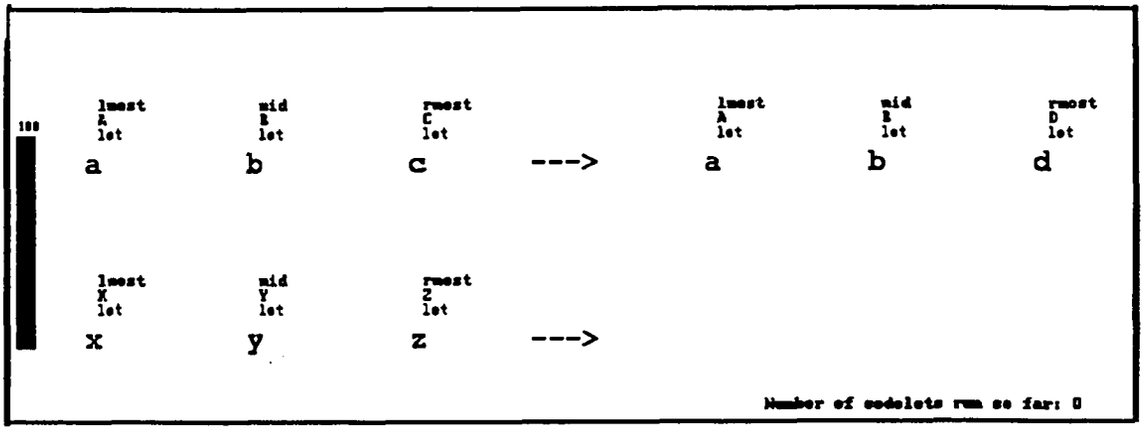
abc ⇒ abd, xyz ⇒ ?

The set of screen dumps given below shows one way in which Copycat arrives at the answer **wyz** after hitting the impasse brought on by its inability to take the successor of *Z*. The two main mechanisms for resolving the impasse are (1) raising the temperature, by allowing structures to be broken more easily (by breaker codelets and by rival structures) and allowing less-obvious pathways to have a better chance of being explored, and (2) at the same time focusing attention on the apparent *cause* of the impasse: the **z** in **xyz**.

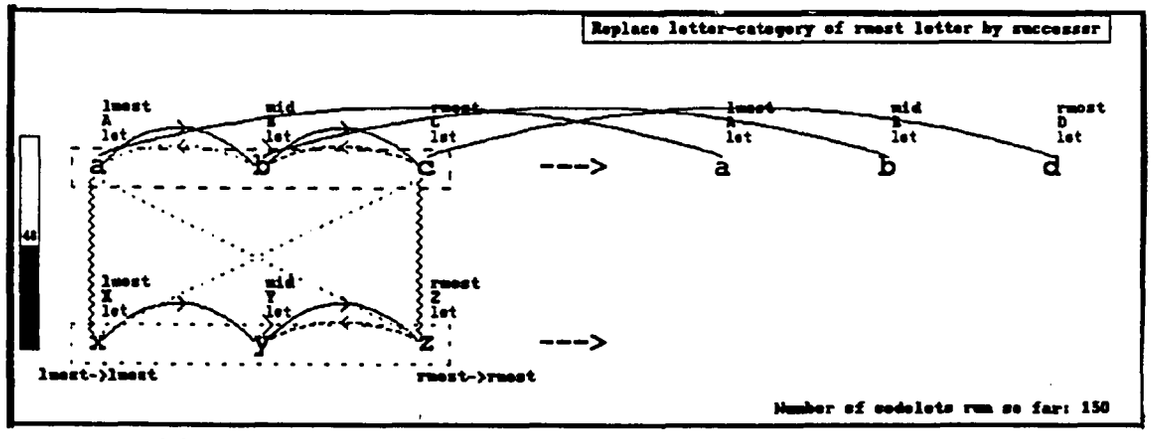
Part of this focusing of attention involves high activation of the node *Z*, which in turn spreads activation to the node *last* (*Z* being the last letter in the alphabet). The activation of *last* greatly increases the probability that it will be attached to the **z** as a description. The node *last* also spreads activation to its neighbor *first*, and this, combined with the fact that *alphabetic-position* is now seen as a relevant way of describing objects, gives *first* a good chance to be attached to the **a**. When this has taken place, a correspondence between the **a** and the **z** (via a *first* ⇒ *last* concept-mapping) is much more plausible, given that the notions of *first* and *last* have been brought into the program's perception of the problem. As was mentioned earlier, concept-mappings that take into account deep similarities (e.g., between *first* and *last*) are seen as strong, but this pressure conflicts with a resistance to making slippages between deep aspects of the two situations. The idea is that there should be a desire to avoid slippage as much as possible, since a perfect analogy is one in which no slippages are needed at all (e.g., "**abc ⇒ abd, ijk ⇒ ijl**"). If one is forced to make slippages, then the more shallow the descriptions that slip, the better, since in making an analogy, one wants to preserve the *essence* of the two situations, which means that deep aspects should remain invariant. However, a good analogy should *expose* deep aspects of the two situations that might not have been recognized before, in the way that the *first* ⇒ *last* concept-mapping exposes a deep similarity between **abc** and **xyz**. Thus in analogy-making there is a fundamental conflict between a resistance to deep slippages and a desire for deep concept-mappings.

The upshot is that in Copycat, it takes strong pressures (including high temperature, which increases the chances of low-probability, risky slippages) to force the *first* ⇒ *last* slippage, but once it has been made, it is seen as quite strong, and its strength increases even more when a resolution to the impasse begins to fall into place as a result of it.

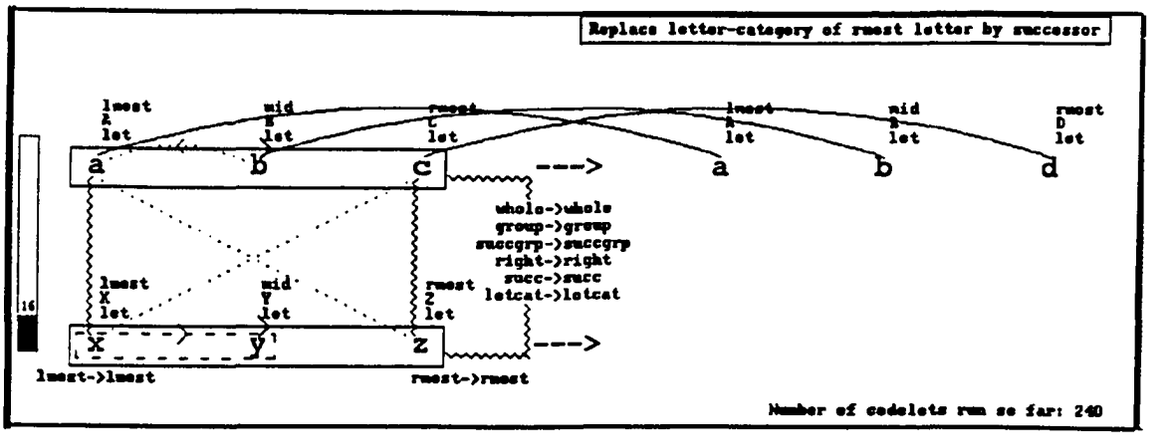
The **a–z** correspondence has to fight against much of the currently existing structure, but if it can prevail (and this is more likely at high temperature), it can trigger a complete restructuring of the program's previous perception of the strings: the strings **abc** and **xyz** can be seen as opposites in both spatial and alphabetic direction, with the **c** corresponding to the **x**. This view leads to the slippages *rightmost* ⇒ *leftmost* and *successor* ⇒ *predecessor*, causing the program to translate the original rule as "Replace *leftmost* letter by *predecessor*", yielding the answer **wyz**.



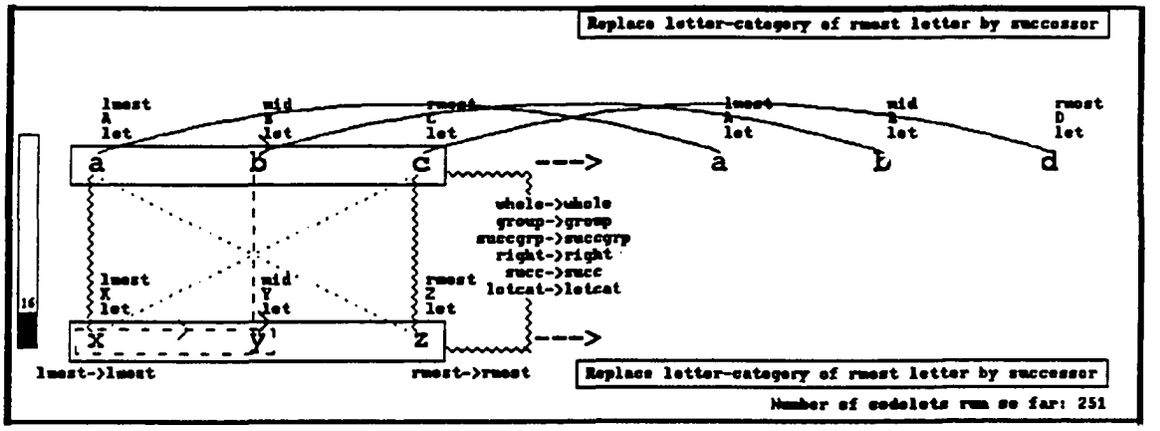
1. The program is presented with the three strings. Descriptions are displayed again here because there will be important additions to them in the course of the run. The Slipnet is also displayed in some of the frames.



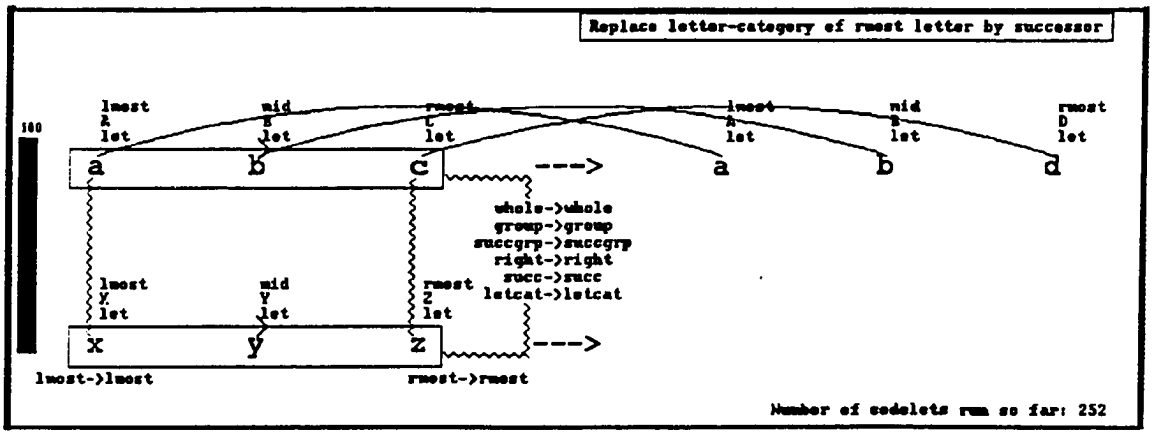
2. 150 codelets into the run, everything is proceeding well, similarly to the run on "**abc** ⇒ **abd**, **ijk** ⇒ ?". There are two strong vertical correspondences (*leftmost* ⇒ *leftmost* and *rightmost* ⇒ *rightmost*). Diagonal correspondences (*leftmost* ⇒ *rightmost* and *rightmost* ⇒ *leftmost*) are being fleetingly examined, but as in "**abc** ⇒ **abd**, **ijk** ⇒ ?" they are very weak and of very low priority for further examination.



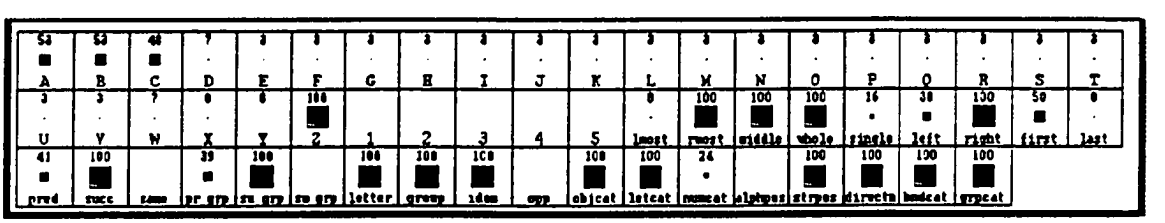
3. After 240 codelets have run, things have been structured just as in the run on "abc ⇒ abd, ijk ⇒ ?", the temperature is very low, and the program is almost ready to try to construct its answer.



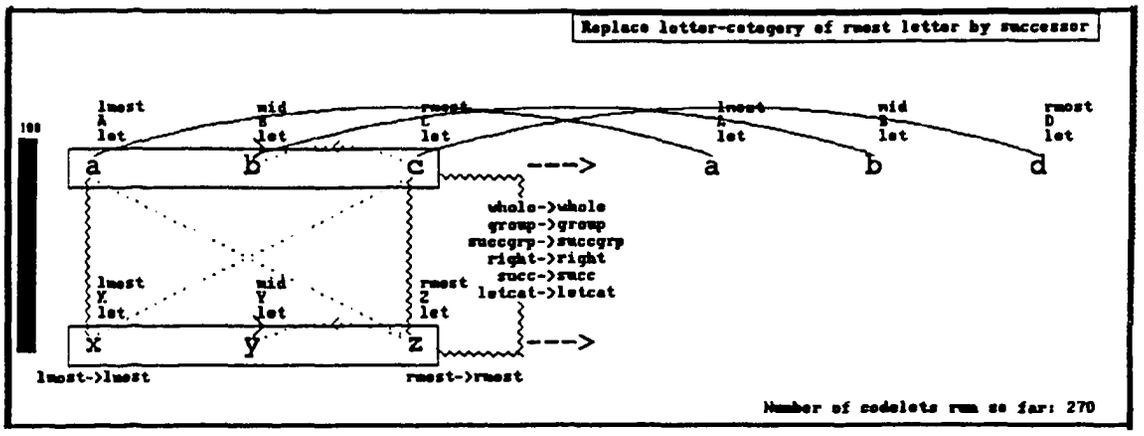
4. Since no slippages are involved in the correspondences between abc and xyz, the rule needs no translation. As it did for ijk, the program attempts to follow the rule "Replace letter-category of rightmost letter by successor", but hits a snag: Z has no successor.



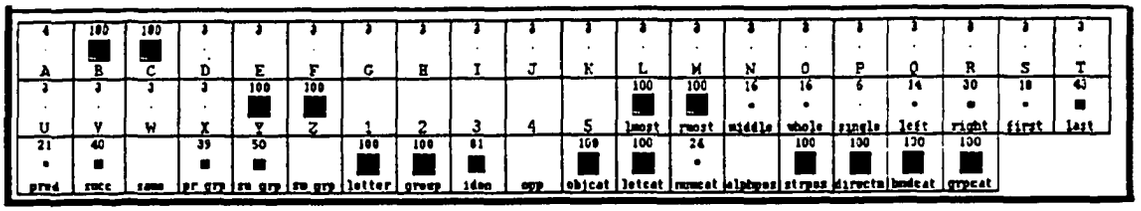
5. **Workspace:** In response to the snag, the temperature shoots up from 16 to 100, reflecting the fact that the program is now at an impasse and that it has gone from being very certain about the quality of the structures it has built to being quite uncertain and far away from an answer. The temperature is clamped at 100, reflecting a "state of emergency" which will not be revoked until the program judges that progress (in the form of new structures) has been made. At this high temperature, actions that normally have a low-probability of occuring (e.g., breaker codelets succeeding in breaking structures) are more likely to take place. But even at this maximum temperature, decisions are not totally random; the same kinds of biases exist at high and low temperatures. The biases just become more and more pronounced as the temperature falls.



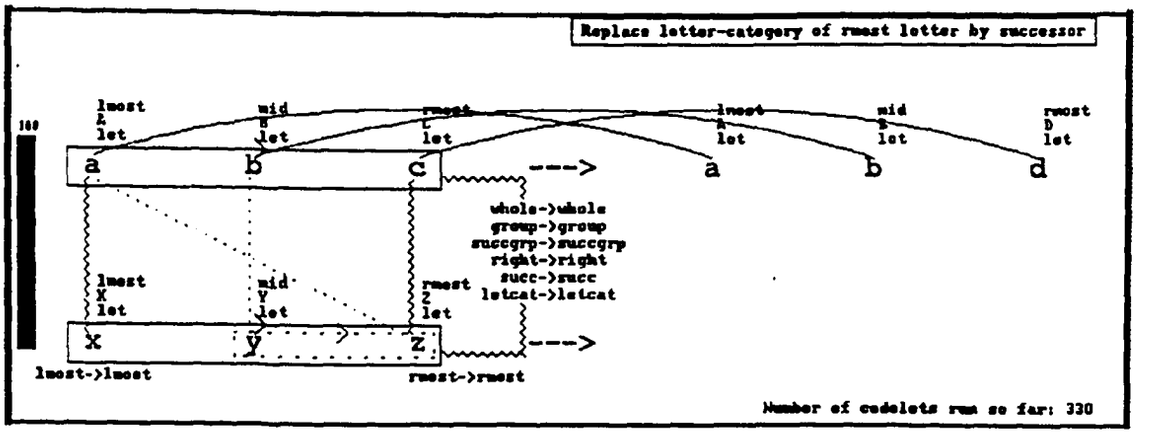
Slipnet: The program's other response to the impasse is to focus on its apparent cause: the **z**. It does this by clamping the activation of all of the z's descriptors at 100, thus making the **z** very salient and making these descriptors a strong focus of attention. Notice in the Slipnet that the nodes *Z*, *rightmost*, and *letter* are all fully activated. As with temperature, these clamps will not be released until the program determines that a sufficient amount of progress has been made towards getting out of this impasse. That is, every time the Slipnet is updated, the program checks to see if any *new* structures have been built, and if so, decides probabilistically, based on their strength, whether to rescind this "state of emergency".



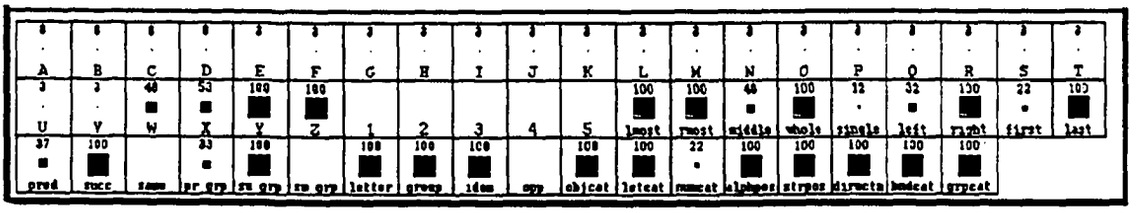
6. **Workspace:** Several possible new structures are being considered, but none have yet gotten very far, and the original structures remain intact.



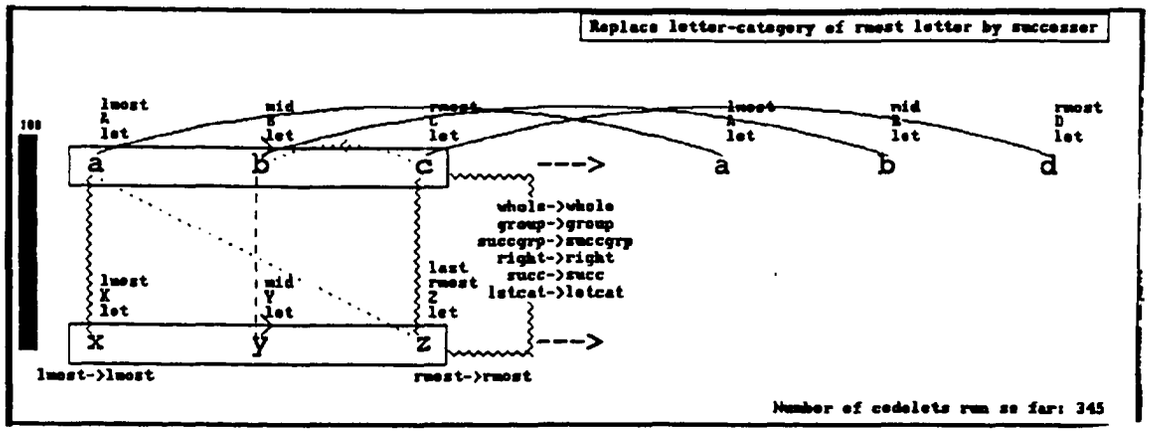
Slipnet: The node *Z* remains active and is spreading activation to the node *last*, which is beginning to become activated.



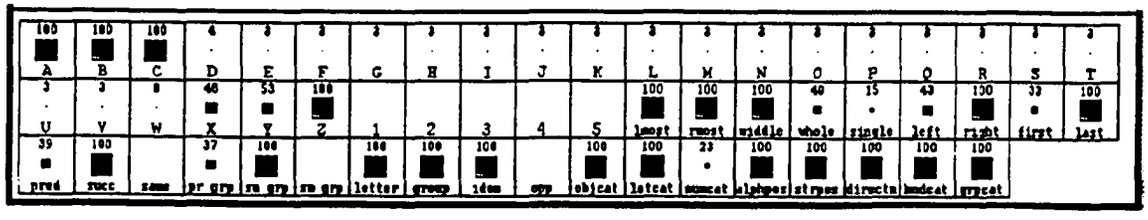
7. **Workspace:** The Workspace is still stuck in the original state. The proposed a–z correspondence (here lacking the *first* ⇒ *last* concept-mapping) is too weak to have much chance of going anywhere, even at this high temperature.



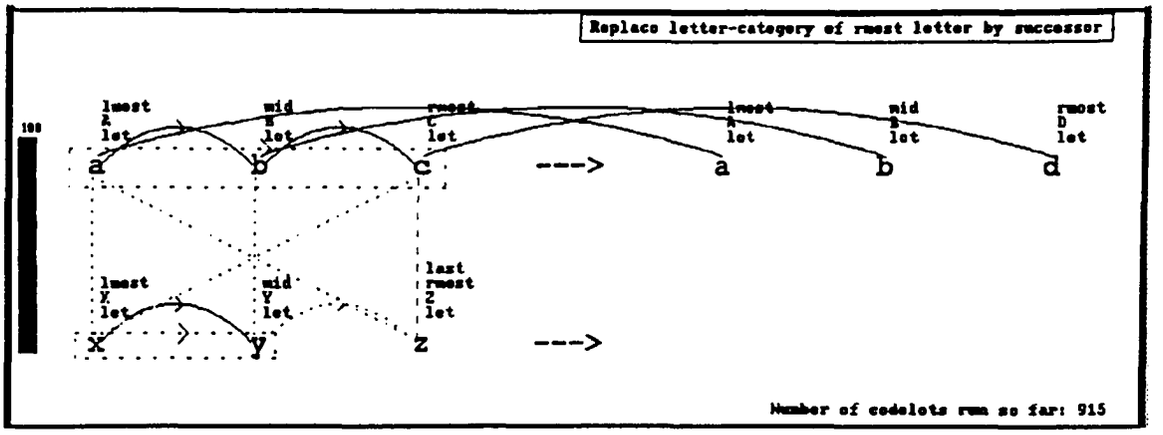
Slipnet: The node *last* is now fully active, and has spread activation to *first* and to *alphabetic-position*, which is also now fully active and is posting top-down description-scouts to try to make descriptions of this type.



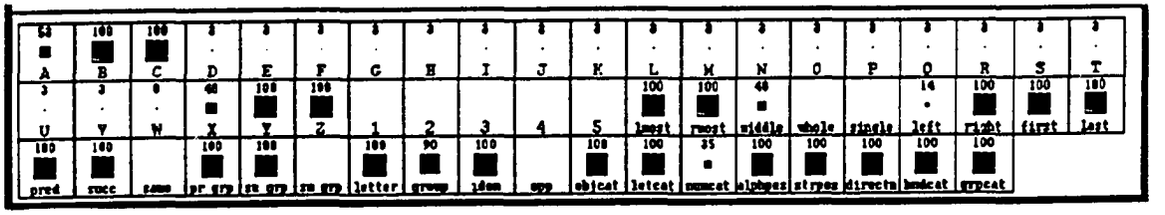
8. **Workspace:** After 345 codelets have run, *last* has been added to the z's list of descriptions as a result of top-down pressures from the activation of *last* and *alphabetic-position.*



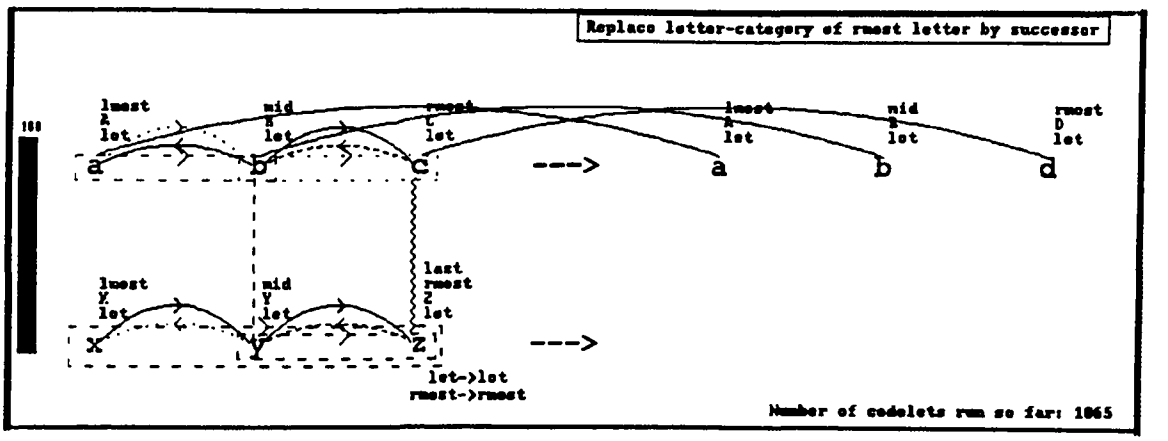
Slipnet: In the Slipnet, the node *first* continues to gain activation from *last.*



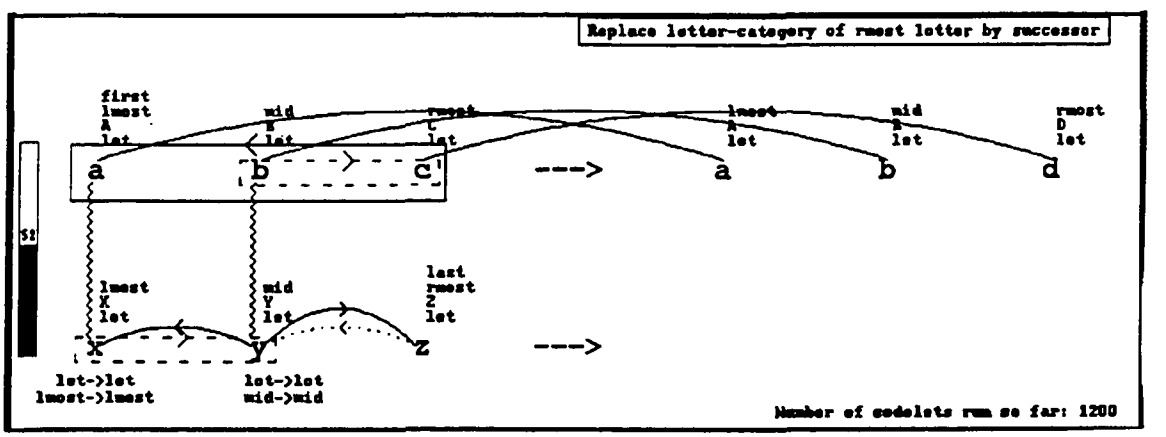
9. **Workspace:** 915 codelets into the run (570 codelets after the previous screen dump), after much thrashing by the program and little progress, several breaker codelets have succeeded in breaking some structures, though the skeleton of the original successorship structure is still intact. (Breaker codelets have a good chance of running and breaking structures only at high temperatures.) As was detailed in Chapter 3, both bottom-up and top-down codelets continue to be posted to the Coderack as the program runs. What have they all been doing? Most of the codelets that run are redundantly working on building the same structures that already exist. This redundancy is an essential part of the program; it allows *statistics*, rather than any single codelet or small set of codelets, to control what happens on a large scale. Other codelets are trying (and so far, failing) to build new structures, and yet others are attempting (and occasionally succeeding) to break existing structures. A large number of the codelets are focusing again and again on the **z**, which is very salient, now having *four* fully active descriptors. Often at this point what happens is that the rule is broken and the weaker rule "Replace letter-category of rightmost letter by *D*" is built (this is more likely than usual, due to the high temperature and lack of progress on other fronts), and is used by the program to get the answer **xyd**.



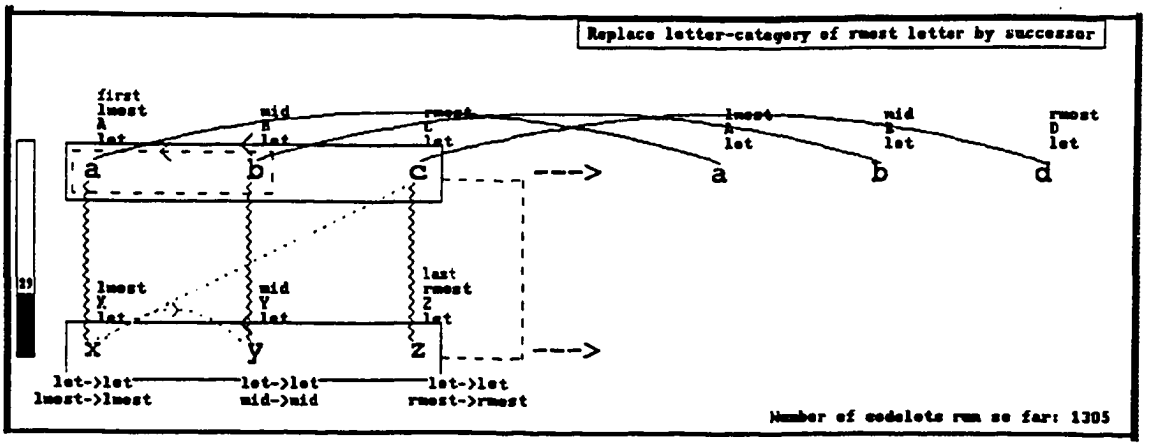
Slipnet: The node *last* has continued to spread activation to *first*, which is now fully activated.



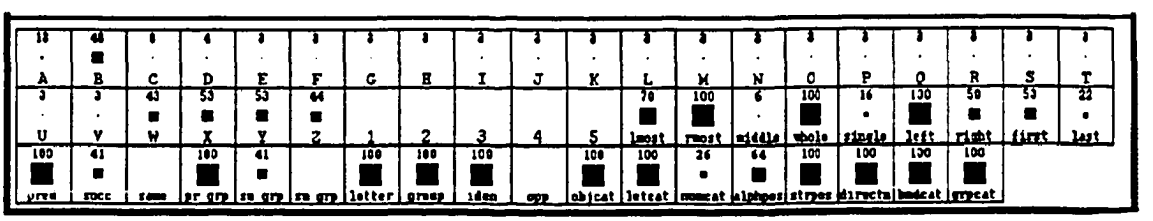
10. At codelet-step 1065, a bit of restructuring is being tried out: a b–a predecessor bond has broken the a–b successor bond.



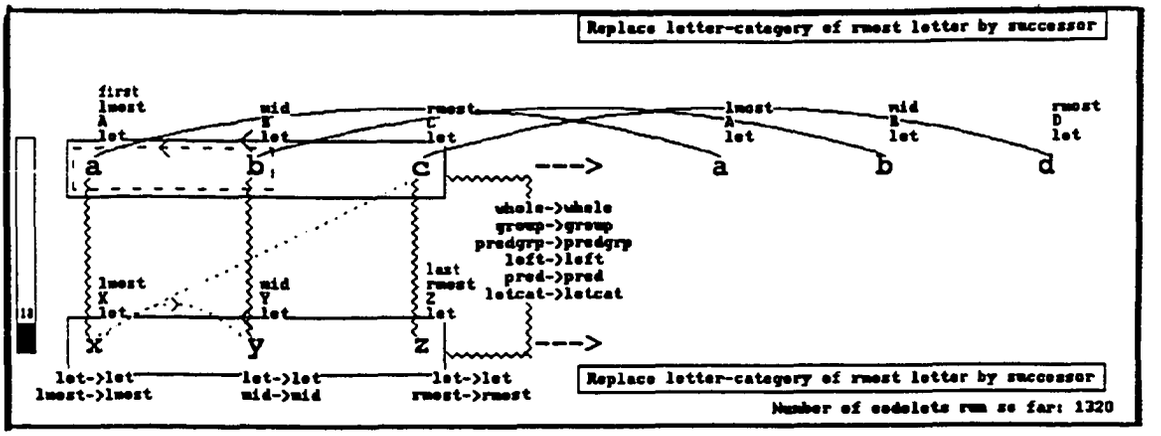
11. The program's view of both strings is in the midst of being restructured, gradually changing from right-going successorship to left-going predecessorship. In the hopes that this is a promising new course, the program has released the clamp on the temperature, which has fallen to the value indicated by the estimated quality of the existing structures (here, 52). Meanwhile, the descriptor *first* has been added to the a's list of descriptions.



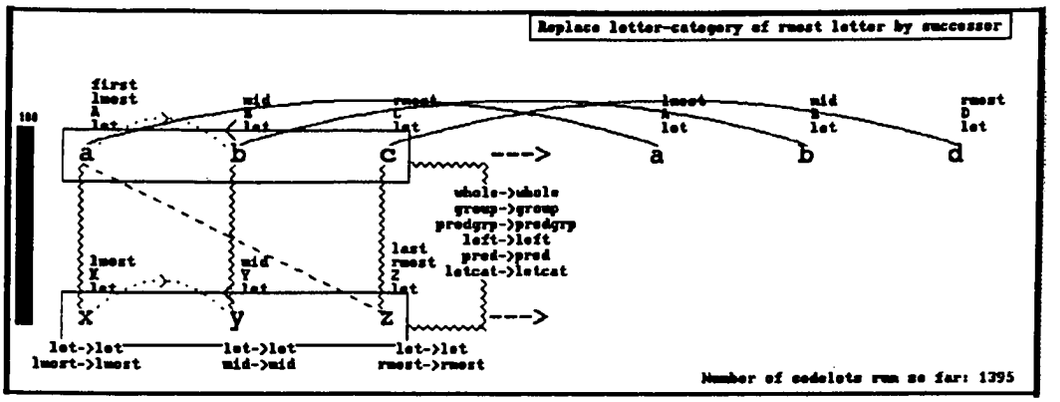
12. **Workspace:** A new viewpoint has taken over, with both strings now grouped as left-going predecessor groups, and with the low temperature reflecting the program's high assessment of this new way of structuring things.



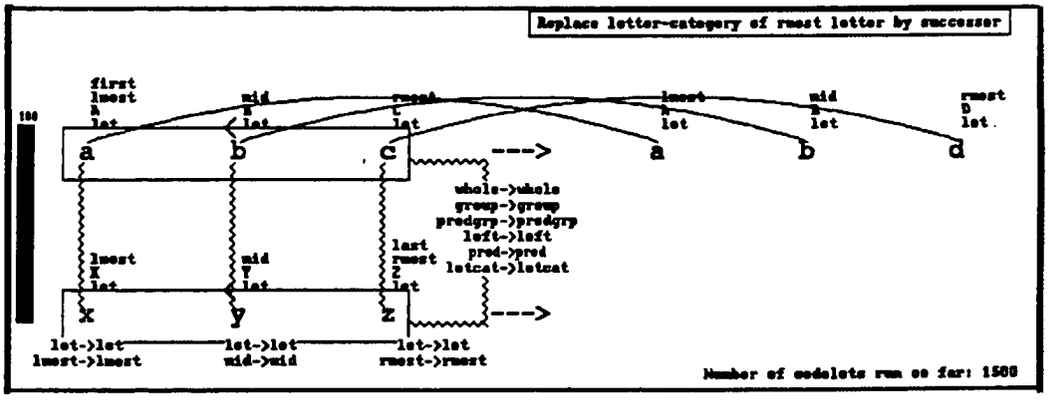
Slipnet: The nodes *first*, *last*, and *alphabetic-position* have decayed considerably, and these descriptors are thus no longer relevant (indicated by the fact that the descriptors are no longer in boldface), and are now being ignored.



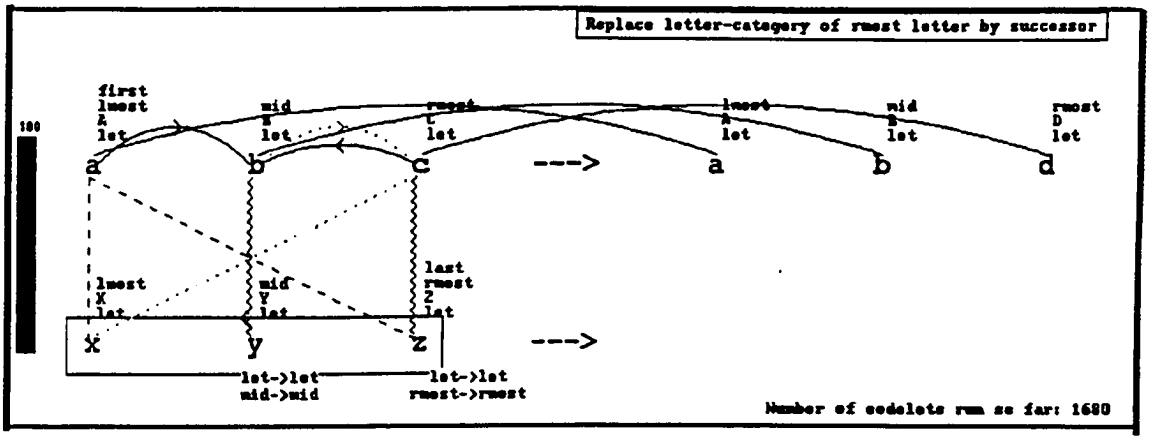
13. The program has used this new, seemingly good set of structures to attempt once again to get an answer, but surprise, surprise: the same snag appears again. One could certainly say that the program shows a lack of common sense for having expected that this trivial form of restructuring could resolve the impasse. But it could also be said that people often get pulled into mental dead-end paths whose futility should have been obvious in the first place. Once one gets started along a certain mental pathway, it is sometimes hard to avert it; obvious ways of viewing situations (such as trying to take the successor of *Z* here) act like attractors; it is hard to avoid them. In general, this is a *useful* feature of perception, because in real life, the most obvious view is usually the right view, so it is good to be quickly drawn into it. However, in some situations, this results in behavior like that of the program on this problem, where you are drawn again and again into the same wrong way of looking at things, perhaps with slight variations. Unfortunately, this happens to the program far too often; as will be discussed in the next section, during an average run on this problem, Copycat continually gets into states that cause it to hit the same snag over and over again (on average 9 times before getting an answer), because it lacks some essential mechanisms for remembering and watching its own behavior. The need for such "self-watching" mechanisms will be discussed further in Chapter 6.



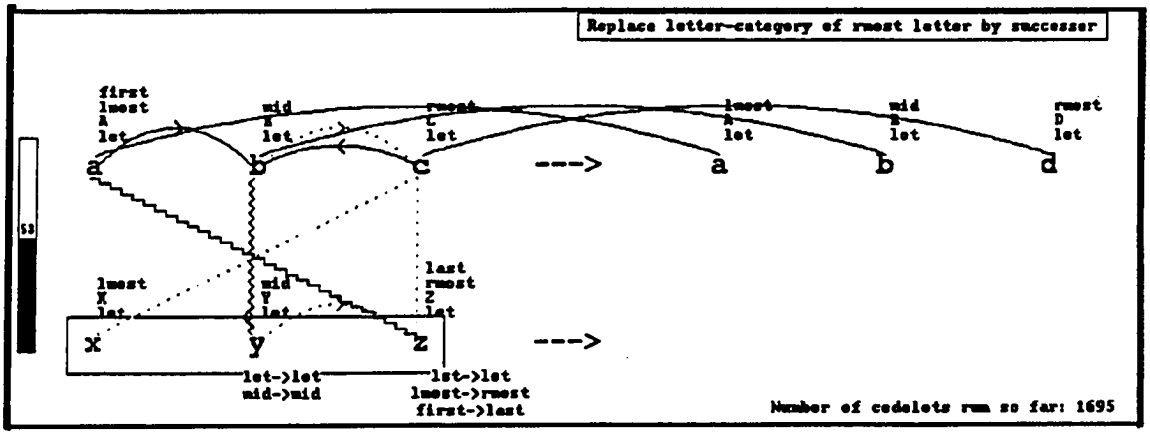
14. Now the program is back at "square one". The temperature has shot up again and the z's descriptors have again been clamped in response to the snag. The a and the z are now both quite salient (since they each have four fully active descriptors), and are thus being chosen very often by codelets. Now a correspondence between them is more plausible, because of the possibility of the concept-mapping *first* ⇒ *last*. As was pointed out earlier, it is initially difficult to make this slippage because of the conceptual depth of the descriptors involved (deep slippages are harder to make than shallow ones), but once it is made, it is seen as fairly strong (deep concept-mappings are stronger than shallow ones). An a–z correspondence has been proposed and has passed its strength test (thanks in part to the high temperature, which makes intrinsically unlikely events more likely), but it still faces a lot of competition from the still quite strong currently existing structures. Note that even at high temperature and in this desperate condition, it is still essential for the program to resist unusual notions—they should be allowed to be seriously considered only under strong pressures. Otherwise the program would be wasting all its time exploring unmotivated crazy possibilities.



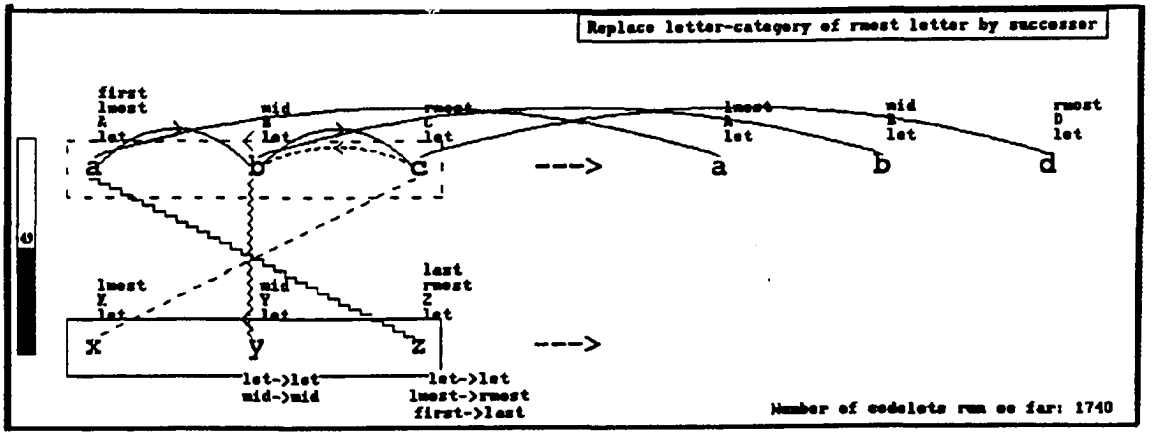
15. The a–z correspondence did not manage to defeat the existing rival structures, and it has fizzled.



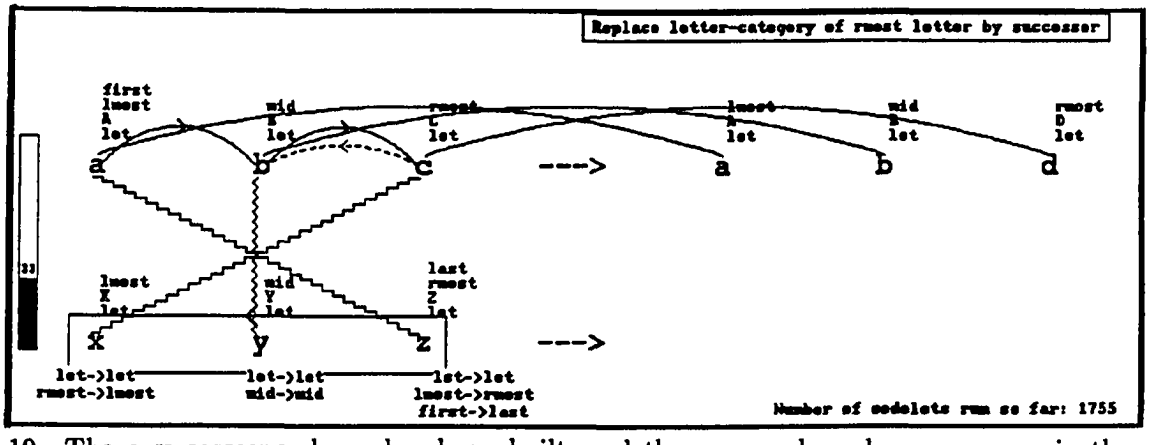
16. Some structures have been broken, and a bit of restructuring is being tried again: this time an **a**–**b** successor bond defeated the **b**–**a** predecessor bond. Another attempt is being made to build the **a**–**z** correspondence (the **a** and **z** remain quite salient, so many attempts are being made to use them in structures), but it still faces strong competition from the existing **c**–**z** correspondence.



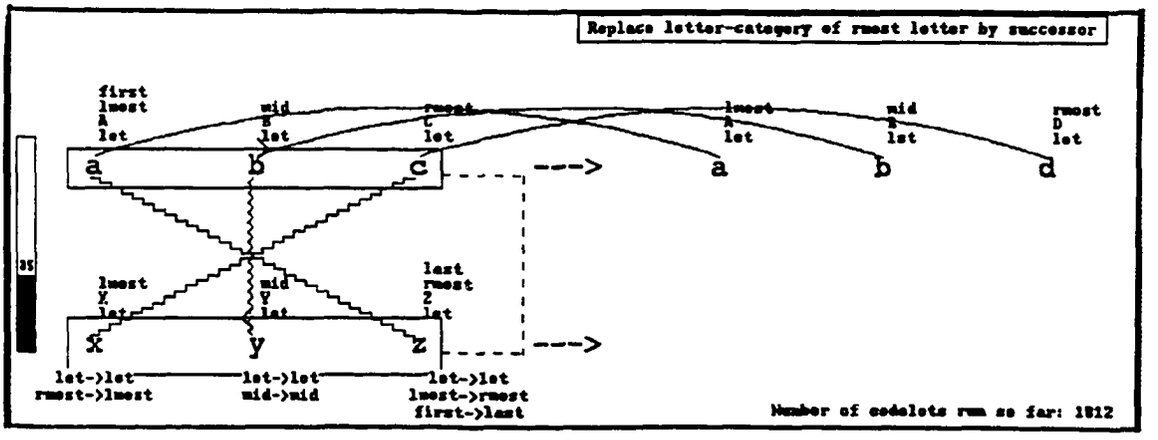
17. Thanks to the combination of the strength of the *first* ⇒ *last* concept-mapping, high temperature, and also to statistics (that is, a large number of tries), the **a**–**z** correspondence has beaten the normally far stronger **c**–**z** correspondence (though the latter is being considered again). The creation of this fairly strong new structure has caused the temperature to be unclamped. The new correspondence has two slippages: *leftmost* ⇒ *rightmost* and *first* ⇒ *last*, and in response to these instances, the node *opposite* has suddenly jumped into prominence. The existence of the **a**–**z** correspondence and the activation of *opposite* will make the proposed **c**–**x** correspondence (before, too weak to have much of a chance at all) much more plausible.



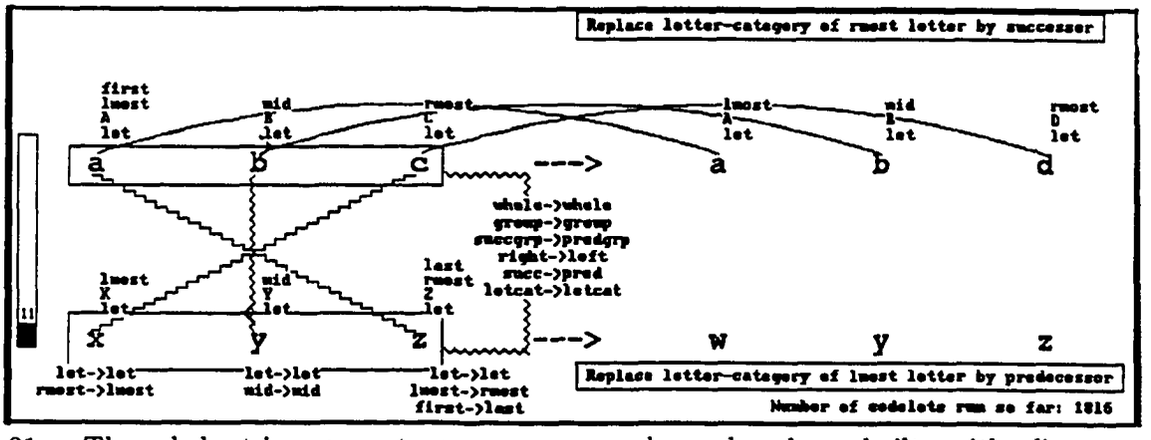
18. The competing proposed c–z correspondence has fizzled. Also, partially in response to the new diagonal correspondence, the initial string is being viewed as consisting of right-going successor bonds—the opposite of the bonds in the target string. However, there is still some competition lurking in the form of a proposed group threatening to turn the whole initial string around so that it is in the same direction as the target string. The c–x correspondence has passed its strength test and is waiting to be built.



19. The c–x correspondence has been built, and the proposed predecessor group in the initial string has fizzled. The temperature has fallen to 33, reflecting the estimated promise of these new structures.



20. The initial string has now been grouped as a right-going successor group, opposite to the target string. This turnaround was made possible by the diagonal correspondences. There is a whole-string group-to-group mapping being considered.



21. The whole-string group-to-group correspondence has been built, with slippages *successor-group* ⇒ *predecessor-group*, *right* ⇒ *left*, and *successor* ⇒ *predecessor*. The last, along with the slippage *rightmost* ⇒ *leftmost*, is used to construct a sweeping translation of the rule: "Replace letter-category of *leftmost* letter by *predecessor*", yielding the answer **wyz**.

4.4 Summary

The series of screen dumps presented in this chapter have hopefully given the reader a better idea of how all the mechanisms described in Chapter 3 work together to produce a system that can flexibly adapt its concepts to new situations that it is presented with. As can be seen from the screen dumps, Copycat starts from a standard initial state on each new problem, but as the program runs, it discovers unique aspects of the problem, bringing out certain associations while downplaying others, allowing it (usually) to home in on a suitable set of relevant concepts and avenues of approach. In addition, when the system's original approach leads it to an impasse, it is able to fluidly restructure its perceptions to find a better way of looking at things.

The screen dumps have also hopefully made clearer the fundamental roles of nondeterminism, parallelism, non-centralized and simple perceptual agents (i.e., codelets), the interaction of bottom-up and top-down pressures, and the reliance on statistically emergent (rather than explicitly programmed) high-level behavior in achieving these abilities. The claim being made for this model is that these are also fundamental features of high-level perception in general.

The result of all these features is an emergent parallel terraced scan of possibilities, in which a fight for cognitive resources takes place, and in which one point of view gradually (or sometimes rapidly) comes to dominate. Nondeterminism pervades this process. Large, global, deterministic decisions are never made (except perhaps towards the end of a run). The system relies instead on the accumulation of small, local, nondeterministic decisions, none of which alone is particularly important for the final outcome of the run. As could be seen in the screen dumps, large-scale effects occur only through the statistics of the lower levels: the ubiquitous notion of a "pressure" in the system is really a shorthand for the statistical effects over time of a large number of codelets and of activation patterns of nodes in the Slipnet.

The program starts out exploring possible structures with a high degree of randomness, and lets both *a priori* biases and information accumulated along the way guide the evolving search. The idea of the parallel terraced scan is to try to allocate time to different paths of exploration in proportion to their estimated promise. As is illustrated by the two-armed bandit problem discussed earlier, it is a bad idea to devote all of one's resources to what currently seems to be the best path if one has very little information on which to base one's estimate of quality. It would also defeat the purpose of the parallel terraced scan if the

promise of every single possibility had to be evaluated before any further exploration could be done—there are too many possibilities to be evaluated. The best strategy is to explore many different possibilities (without excluding any *a priori*), continuously adjusting the speed of exploration of each possibility as a function both of moment-to-moment estimates of its promise as it unfolds and of the global sense of how *reliable* those estimates are. In Copycat this effect is an emergent one, achieved statistically though a large number of temperature-controlled nondeterministic choices.

As was seen in the screen dumps, as structures are formed and a global interpretation coalesces, the system gradually makes a transition from being quite parallel, random, and dominated by bottom-up forces to being more deterministic, serial, and dominated by top-down forces. We believe that such a transition is characteristic of high-level perception in general.

CHAPTER V

COPYCAT'S PERFORMANCE ON VARIANTS OF THE FIVE TARGET PROBLEMS

5.1 Introduction

In this chapter I present the performance of the program on 27 variants of the five target problems. As the previous chapter demonstrated, Copycat models how various pressures interact, compete, and are resolved in the process of interpreting situations and making analogies between situations. As will be seen, the variants given here constitute *families* of analogy problems that explore in greater detail certain of the issues in perception and analogy-making that have been discussed in this dissertation. Copycat's behavior on these problems demonstrates how it deals with these issues, how it responds to variations in pressures, and how it is able to fluidly adapt to a range of different situations (starting from exactly the same state on each new problem).

There are a huge number of ways in which the original five problems can be varied. For example, consider "**aabc** ⇒ **aabd**, **ijkk** ⇒ ?", a variant of "**abc** ⇒ **abd**, **ijk** ⇒ ?", in which the doubling of letters is meant to alter the "stresses" on various locations in the strings **abc** and **ijk**. One effect this might have is to make the **a** and the **k** more salient and more similar to each other, thus pushing towards a diagonal mapping in which the two double-letters are seen to correspond. Another variation would be to triple the letters instead of doubling them, which would again slightly alter the pressures, perhaps increasing the salience of the sameness groups. Many other variations in this vein could be made as well.

Another way of manipulating pressures is to include distinguished letters—**a** and **z**—in strategic spots, since it is possible that they will be seen as more salient than other letters and thus attract more attention, changing the pressures in the problem.

Another technique is to alter the relational fabric of a string or of a segment of a string—specifically, to use successor relations where sameness relations existed, or vice versa. A variant of this technique is to get rid of a fabric altogether, or to introduce a fabric where there originally was none. Yet another technique is to experiment with strings of different lengths.

Another very important technique is to manipulate pressures by introducing or deleting same-category letters. These kinds of variations were illustrated in problems 1*a*–*d* in Chapter 2. For example, given the change **abc** ⇒ **abd**, the target **cde** is similar to the target **ijk** except that it contains a **c**, which might attract special attention because of its identity with the **c** of **abc**. By including more or fewer such letters, or by manipulating their positions inside the strings, one can create a vast spectrum of differing pressures.

Each of these pressures taken singly can provide a wealth of variants on a given problem, but when several of them are used in conjunction, one can create a gigantic family of problems forming a vast halo surrounding an original problem. This chapter simply surveys a small sampling of such variants on the five target problems, revealing how the variations in pressures affect the program's behavior.

Each variant highlights and tests some aspect of the program's behavior, and the sum total of all these results gives a clear picture of the program's abilities, thus further addressing the artificial-intelligence criteria discussed in Chapter 2. The results in this chapter also give a sense of what kinds of answers Copycat tends to prefer. A bit anthropomorphically, these results can be said to illustrate the program's "personality".

Even though the variants are divided into five sets corresponding to the five target problems, in many cases the division is somewhat arbitrary, since many of the problems could be considered variants of more than one of the five original problems.

The results are presented in the form of bar graphs similar to the ones given in the previous chapter. Copycat was run on most of the problems 200 times (except in a few special cases, as explained below)—enough runs to get a set of reliable statistics for the program's behavior on each problem (which can then be compared with the results on the original problem), though in some cases not enough runs to get instances of some of the rarer "fringe" answers that came up on the 1000-run results given in the previous chapter

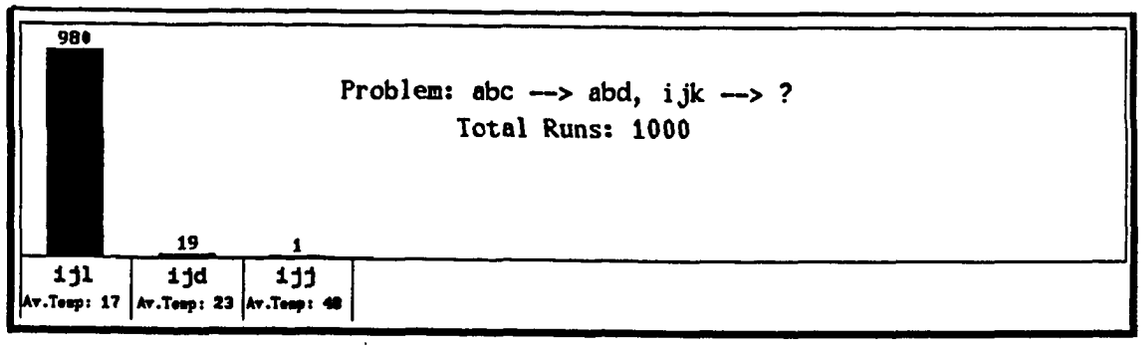
(though many fringe answers do appear). The bar graph for the appropriate one of the five target problems is displayed again at the beginning of each section, so that it can be referred to more easily.

Most of these variants were included in the surveys I gave to people, and the results for each problem are given here along with the discussion of Copycat's performance on that problem. For some of the problems given to people in the survey, an attempt was made to reduce the influence from previous problems by giving a version with different letters when the letter-categories made no difference (e.g., "**bcd ⇒ bce, xlg ⇒ ?**" was given instead of "**abc ⇒ abd, xlg ⇒ ?**"); for clarity's sake, I translate these back to the original letters when giving the results. Different groups of people were given different sets of problems (also as an attempt to minimize cross-influences among similar problems), so different problems were answered by different numbers of subjects. The results from this survey can be used in two ways: first, as in the previous chapter, to compare the range of Copycat's answers with those of people, and second, to see if the different pressures in the variants affect people in the same ways that they affect Copycat. Of course, such a comparison should be made at the level of general tendencies rather than that of specific frequencies.

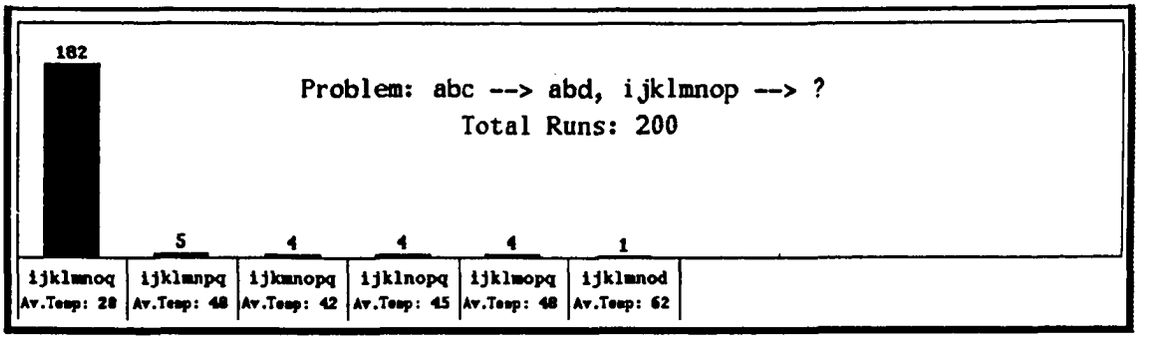
Perhaps the best way to read this chapter and judge Copycat's performance is for readers to try each problem themselves before looking closely at Copycat's answers, and to see how well Copycat did, based on their own judgment of what makes for a good or reasonable set of answers to each problem.

5.2 Variants of "abc ⇒ abd, ijk ⇒ ?"

The bar graph for "**abc ⇒ abd, ijk ⇒ ?**"



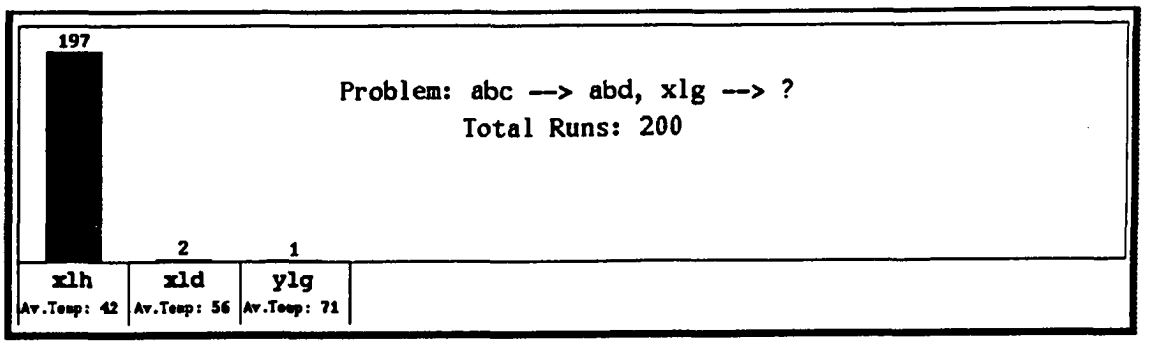
Variant 1: **abc ⇒ abd, ijklmnop ⇒ ?**



In this variant, the length of the target string is extended. This shouldn't affect Copycat's performance on this problem very much: when successor bonds begin to be built in the target string, top-down forces should cause the program to quickly see the entire string as a successor group, just as in the original problem (which I will abbreviate as **ijk**). As can be seen from the bar graph, the proportion of instances of the "Replace rightmost letter by successor" answer (**ijklmnoq**)is almost as high as in **ijk**. The average temperature for this answer is somewhat higher than **ijl** for two reasons: here, there are many letters in the target string that don't correspond to anything in the initial string, and, since the target string is longer, the program doesn't manage to group the whole string here as often as it did in **ijk** (74% of the time during the 200 runs here versus 93% of the time in **ijk**). The bar graph also shows that Copycat gets a variety of low-frequency fringe answers, reflecting various parts of the target string that were grouped and seen as corresponding to the "rightmost letter". All the fringe answers except **ijklmnod** have this property. As will be discussed in the next chapter, this is one of the program's problems: when given long strings, it occasionally makes small groups and does not merge them together into larger groups.

In the survey on people, 10 subjects answered this problem, and the results were much the same as those on the original problem, with 8 **ijklmnoq**'s, 1 **ijklmnod**, and also 1–3 instances each of a few answers involving descriptions Copycat cannot make, such as "third letter" or "rightmost letter of each group of three".

Variant 2: **abc ⇒ abd, xlg ⇒ ?**

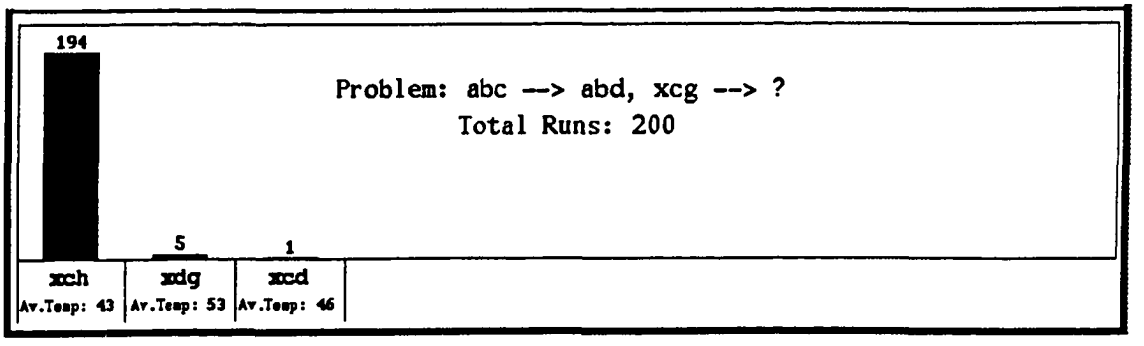


In this variant, there is no relational structure in the target string. As can be seen from the bar graph, this results in higher average temperatures for all the answers, but the "Replace rightmost letter by successor" answer (**xlh**) still wins by a landslide, since this rule is very strong compared to "Replace rightmost letter by *D*" or "Replace *C* by *D*", and triumphs even at these relatively high temperatures. The answer **xld** has roughly the same proportion of instances here as **ijd** had in the original problem. Here there is also a strange answer, **ylg**, that came from a *rightmost* ⇒ *leftmost* mapping between the **c** and the **x** (strange answers like this one are more likely here than in **ijk** because in this problem the temperature tends to stay much higher than it does in **ijk**).

Even though the average temperatures are higher in this variant, there is not a large difference between the time taken by Copycat to get an answer to this problem and to **ijk**: the average number of codelets run in **ijk** was 290, and here, 332. This is because (as was described in Chapter 3), when the program "senses" that there is probably no structure to be found, as in the target string here, it is more willing to give up and to give an answer even though the temperature is high.

In the survey on people, 10 subjects answered this problem, and the results are again much the same as those on the original problem, with 10 **xlh**'s and 1 **xld**.

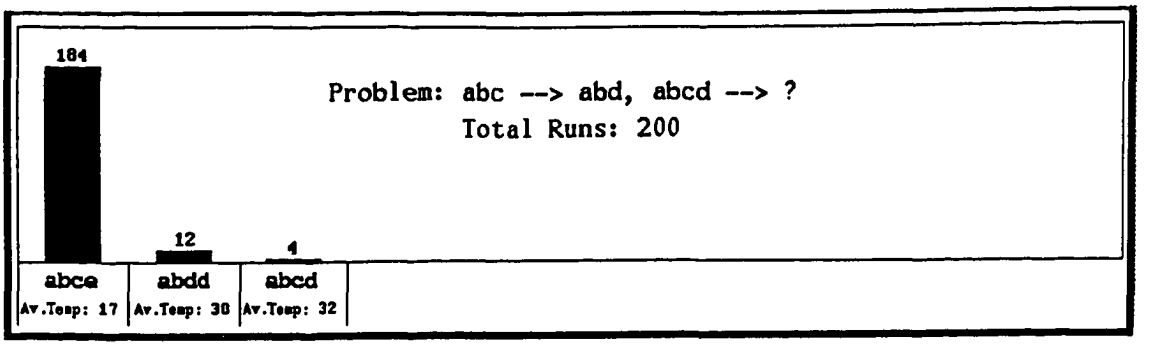
Variant 3: **abc ⇒ abd, xcg ⇒ ?**



This variant is similar to Variant 2, but here the target string contains a **c**, so there should be more pressure than in the original problem to describe the **abc ⇒ abd** change as "Replace *C* by *D*". This pressure did cause the program to construct this rule and get answer **xdg** on five out of 200 runs (this rule was not used even once in 1000 runs on **ijk**, though it is possible in principle). The presence of a **c** in the target string here makes the rule "Replace *C* by *D*" stronger than the rule "Replace rightmost letter by *D*", even though the former rule contains descriptors of lesser conceptual depth (i.e., *C* versus *rightmost*). There are several reasons why the answer **xdg** does not show up even more often: (1) the *C* ⇒ *C* correspondence, though fairly strong, is still quite a bit weaker than the *rightmost* ⇒ *rightmost* correspondence (since the latter has greater conceptual depth); (2) in addition, the *C* ⇒ *C* correspondence has to fight against the strong "Replace rightmost by successor" rule, which creates pressure for the **c** in **abc** to correspond to the rightmost letter of **xcg** rather than to the **c** in **xcg**; and (3) the *C* ⇒ *C* correspondence prevents the **b** from mapping onto anything in the target string, whereas the *rightmost* ⇒ *rightmost* view allows a correspondence between the two strings' respective middle letters.

In the survey, 33 subjects answered this problem. As was the case for Copycat, the results from people show the pressure of the **c** in the target string: here there were 4 **xdg**'s, reflecting the rule "Replace *C* by *D*", whereas there was only one **ijk** given for the original problem, which I am not even sure reflected that rule. It seems that in this context, people (like Copycat) find "Replace *C* by *D*" to be stronger than "Replace rightmost letter by *D*": the answer **xcd** was given just twice here. The answer **xch** was given 29 times, showing that even under these pressures, most people, like Copycat, still considered the rule "Replace rightmost letter by successor" to be the most immediate.

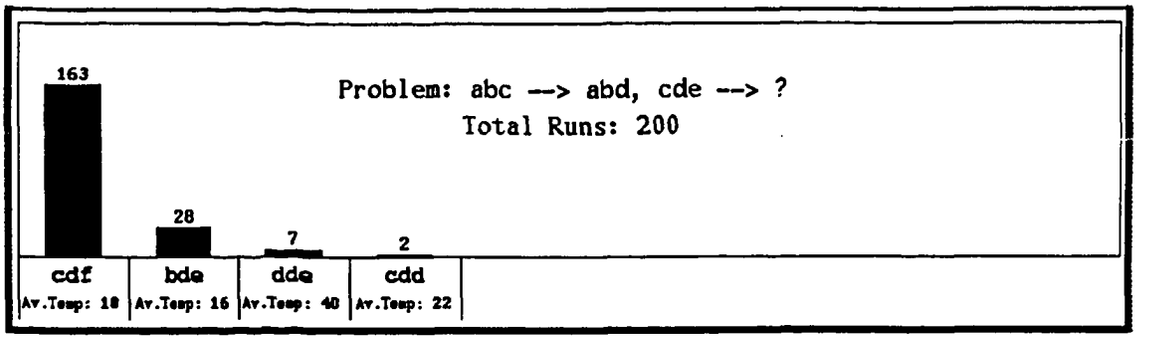
Variant 4: **abc ⇒ abd, abcd ⇒ ?**



Here there is a stronger conflict—the very same three letters appear in the initial and target strings, so it is tempting to map **a** to **a**, **b** to **b**, and **c** to **c**, and to answer **abdd**. This additional pressure for the "Replace *C* by *D*" answer is reflected in the bar graph; it was given 12 out of 200 times, more often than in the previous two variants. But the same pressures discussed for those variants also come up here, as well as the strong pressure to see the initial string and target string map on to each other as wholes, since both can be perceived as successor (or predecessor) groups. The result is a still overwhelming predominance of the "Replace rightmost letter by successor" answer, **abce**. There are also 4 instances of the "Replace rightmost letter by *D*" answer, yielding (coincidentally) a string identical to the target string (Copycat doesn't notice this).

In the survey, 18 subjects answered this problem. As was the case for Copycat, the results from people seem to reveal pressures resulting from the presence of instances of *A*, *B*, and *C* in the target string: here there were 7 **abdd**'s. However, I can't be sure of this, since I don't know whether people were using the rule the rule "Replace *C* by *D*", "Replace third letter by successor", or both (one subject specified both rules for this answer). The "Replace rightmost letter by successor" answer **abce** was (as for Copycat) the most common; it was given 14 times. The answer **abcd** was also given once, though I don't know for sure what the justification was. Two other answers, **abde** and **abef** , were each given once. Both reflect the rule "Replace the third and following letters by their successors", interpreted in different ways.

Variant 5: **abc ⇒ abd, cde ⇒ ?**



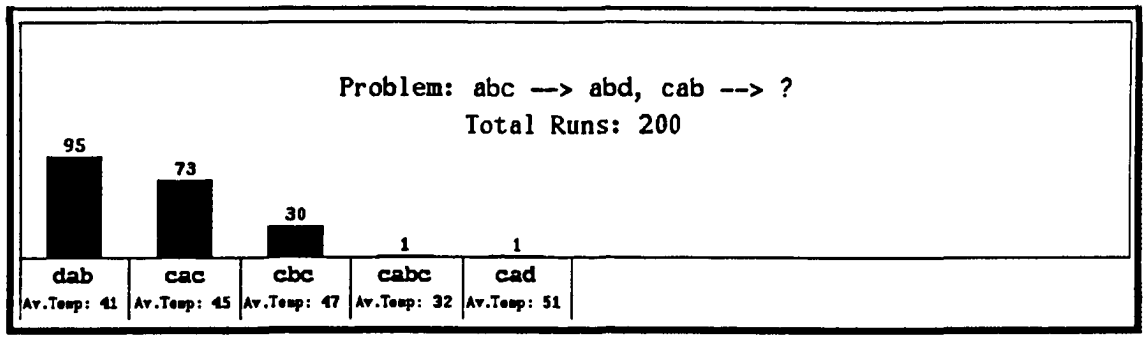
In this variant, as in **ijk**, the target string can be perceived as a successor (or predecessor) group, but there is a C is on the left, which generates a bit more pressure than in the previous variant for the program to make a **c**–**c** correspondence, since it would have both a $C \Rightarrow C$ and a *rightmost* ⇒ *leftmost* concept-mapping. If this correspondence is made, then there is considerable pressure to map the initial string to the target string as a whole, but in opposite spatial and alphabetic directions, yielding answer **bde**. This pressure is reflected in the bar graph: although most of the time the program answers **cdf** ("Replace rightmost letter by successor"), there are also a fair number of **bde** answers. The average temperature for these is roughly the same as for **cdf**: even though *rightmost* ⇒ *leftmost* and *successor* ⇒ *predecessor* slippages have to be made in order to get **bde**, the strength of the $C \Rightarrow C$ mapping balances these slippages, so all in all **bde** is seen as a strong answer. There are also some instances of **dde**, which came either from the rule "Replace C by D", or from the rule "Replace *leftmost* letter by successor" (i.e., the **c**–**c** correspondence was made, but not the whole-string correspondence and therefore not the *successor* ⇒ *predecessor* slippage). Also there were two instances of **cdd**, which came from the rule "Replace rightmost letter by D".

Overall, there are about three times as many $C \Rightarrow C$ answers as in the previous variant, illustrating the stronger pressure here to map the two **c**'s. Many people (including myself) do not feel that **bde** is any more reasonable here than **abdd** was in the previous variant; I think that the program is too willing here to let many things slip for the sake of making the $C \Rightarrow C$ correspondence, and my feeling is that the answer **cdf** should be even more frequent than it is.

Though only a small number of people in the survey answered this problem, the ones who did with me and not with Copycat. They did not appear to feel much pressure exerted by the **C** in the target string: **cdf** was given 10 times, and one person also gave the answer

cdd. This is similar to the results on the original **ijk** problem. No one gave **bde** or **dde**, though I think these answers would come up if there were more subjects in the survey.

Variant 6: **abc** ⇒ **abd**, **cab** ⇒ ?

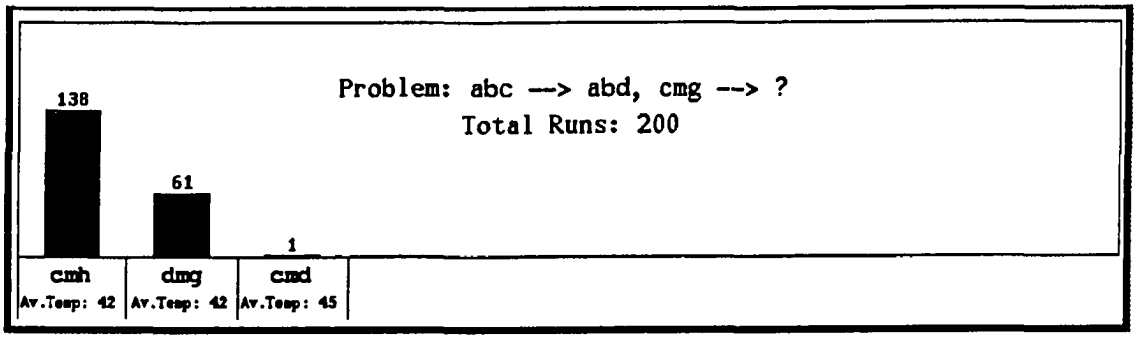


As in the previous variant, the target string here has a **c** on the left, and as in the **abcd** variant, it also contains the same letters as the initial string, so there is some pressure to map the two **a**'s, two **b**'s, and two **c**'s, and the pressure is increased by the additional *rightmost* ⇒ *leftmost* concept-mapping. Moreover, the target string is not in alphabetical order (in either spatial direction), so no whole-group mapping can be made. Thus the $C \Rightarrow C$ mapping in this variant does not face the strong competition from the *rightmost* ⇒ *rightmost* correspondence that existed in the **abcd** variant and in the **cde** variant. In both of those, the same-direction whole-string mapping strongly supported the *rightmost* ⇒ *rightmost* correspondence, but there is no such support here. This can be seen in the bar graph: there are many more instances of the $C \Rightarrow C$ answer (**dab**). However, in spite of these pressures, there are overall still more *rightmost* ⇒ *rightmost* answers than $C \Rightarrow C$ answers: the straightforward **cac**, as well as **cbc**, for which the program grouped the **a** and **b** in the target string and saw this group as the string's "rightmost letter". This shows the strength of the more abstract rule "Replace rightmost letter by successor" over "Replace C by D", even in the face of much pressure for the latter.

There is one instance of the usual "Replace rightmost letter by D" answer (**cad**) and also one instance of a strange answer, **cabc**, which came from a view in which the target string was grouped as **c**–**ab**, the **c** being seen as a group of length 1, and the **ab** a group of length 2. As in **mrrjjj**, the program replaced the length of the rightmost group by its successor, yielding **cabc**, even though the pressures here do not seem to be sufficient to warrant building a single letter group and bringing in the notion of group-length. This happened only once in 200 runs, but I think even that is too often; the program is a bit too willing to perceive single-letter groups.

In the survey, 10 subjects answered this variant, and it seemed to have similar effects on the subjects to those it had on Copycat: there were 5 **dab**'s, 5 **cac**'s, and 1 **cad**. Thus in this context, people (like Copycat) were much more likely to make the $C \Rightarrow C$ mapping than in the previous variant.

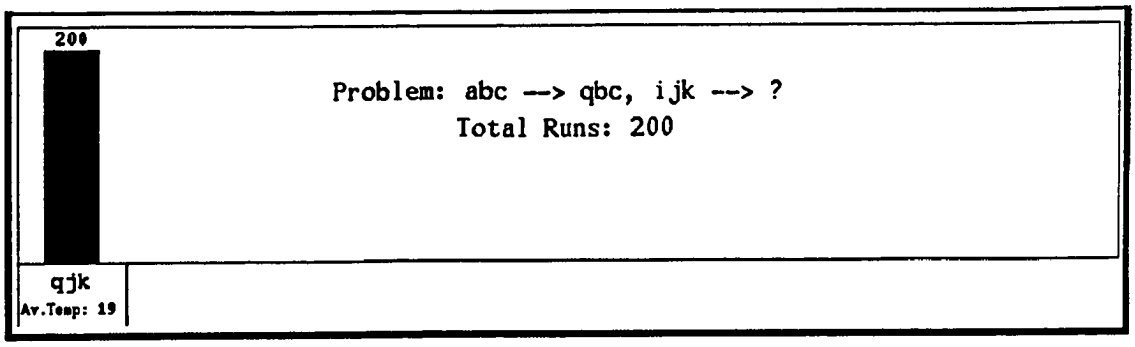
Variant 7: **abc ⇒ abd, cmg ⇒ ?**



Here there is a **c** on the left in the target string (in which there are no relations between letters), but unlike in **cab**, there are no exact letter-category matches for the **a** and the **b**. This increases the pressure to make the *rightmost* ⇒ *rightmost* mapping rather than the $C \Rightarrow C$ mapping. This can be seen in the bar graph, where the $C \Rightarrow C$ answer (**dmg**) is less frequent than in the previous variant.

Eleven subjects answered this problem, and every single one of them answered **cmh**. No other answers were given, though I believe that the other two answers would come up if there were more subjects in the survey. As was the case for Copycat, people felt considerably less pressure here than in the previous variant to map the two **c**'s, though it seems that here it was hardly felt at all by people, whereas Copycat still gave the answer **dmg** fairly often.

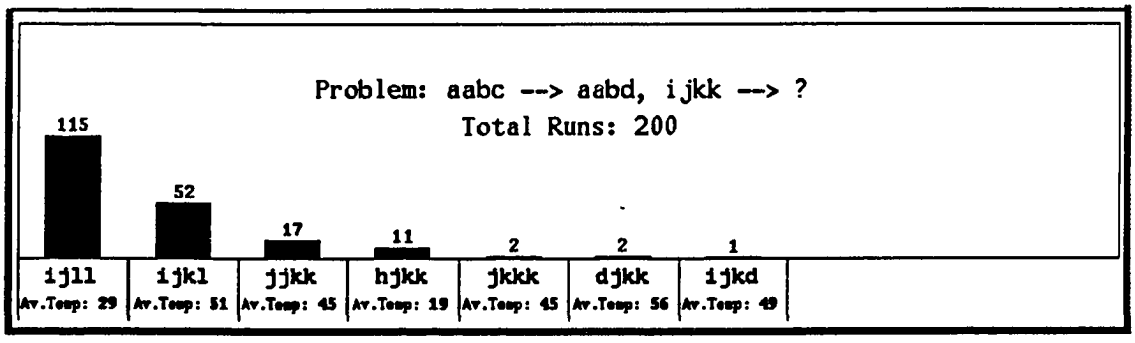
Variant 8: **abc ⇒ qbc, ijk ⇒ ?**



Since *A* and *Q* have no relation in the Slipnet, the possible rules here are: (1) "Replace leftmost letter by *Q*", which yields answer **qjk** (the only answer given by Copycat during these 200 runs); (2) "Replace first letter [of the alphabet] by *Q*", which would yield **ijk** (this rule, never used during the 200 runs, is possible only if the **a** is given the description *first*); and (3) "Replace *A* by *Q*", which would also yield **ijk** (this rule is very weak, and was never used in the 200 runs).

Ten subjects answered this problem. Six answered **qjk**, and the other four gave illegal answers—either answers that involved counting long distances in the alphabet (even though subjects had been instructed not to do so) or answers using the rule "Replace the leftmost letter by any letter".

Variant 9: **aabc ⇒ aabd, ijkk ⇒ ?**

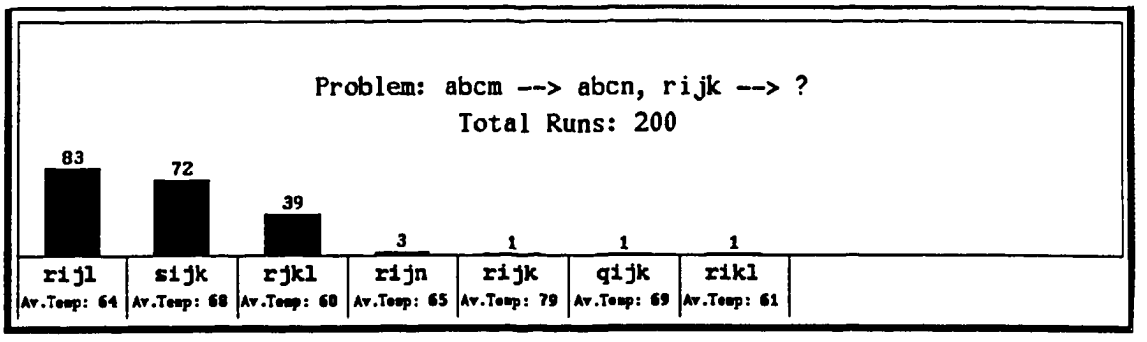


Here there is a double **a** on the left and a double **k** on the right, creating some pressure for the program to see a mapping between the two double-letters, and on the basis of that mapping, to change the leftmost letter **i** instead of the rightmost group **kk** or rightmost letter **k**. The **i** could be changed in two ways: either by replacing it by its successor (**jjkk**) or, based on the diagonal (*leftmost* ⇒ *rightmost*) correspondence, seeing the two strings as going in opposite alphabetic directions and thus replacing the **i** by its *predecessor* (**hjkk**). Even with this pressure to change the **i**, 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 strength of the *leftmost* ⇒ *leftmost*, *rightmost* ⇒ *rightmost* view, even here. However, the pressure is felt to some extent: **jjkk** has a good showing and **hjkk** has some representatives as well (and also has by far the lowest average temperature). This is to be contrasted with the results on **ijk**: in 1000 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 a grouping the *two* leftmost letters), **djkk** (replacing the **i**, but by a **d** instead of by its successor or predecessor), and the usual "Replace rightmost letter by *D*" answer (**ijkd**).

Ten subjects answered this problem. All ten gave the answer **ijkl**. Five other answers were also given, each only once: **ijll**, **jjkk**, **hjkk** (both these "diagonal" answers were given by the same person), **ijlk** (replace third letter by successor), and **ikkk**.[1] The pressure to map **aa** ⇒ **kk** was not strongly felt by the subjects, though this mapping did show up in answers given by one subject (and most likely another subject as well).

Variant 10: **abcm ⇒ abcn, rijk ⇒ ?**



Here, an extra, unrelated letter is added on at opposite ends of the initial and target strings. This creates pressure for the program to map the two successor (or predecessor) groups, **abc** and **ijk**, generating a *leftmost* ⇒ *rightmost* slippage, which in turn generates a *rightmost* ⇒ *leftmost* slippage, lobbying for the answer **sijk**. Copycat gave that answer almost as often as it gave **rijl**, which is based on the straightforward "Replace rightmost

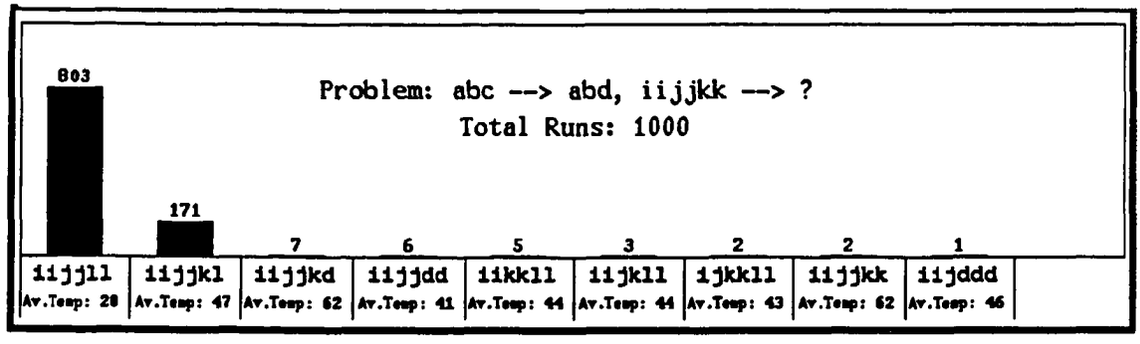
[1] I am not sure what the justification was for **ikkk**, but it very likely was the following: make the diagonal mapping between the groups **aa** to the group **kk**, and the opposite diagonal mapping between the groups **bc** and **ij**. Then map the rightmost letter of **bc** (the **c**) to the rightmost letter of **ij** (the **j**), and replace it by its successor, yielding the answer **ikkk**. The current version of Copycat could not get this answer because it is not able to make descriptions such as "rightmost letter of leftmost group".

letter by successor" rule. It also answered **rjkl** a fair amount of the time, based on seeing the group **ijk** as the "rightmost element" of the target string, and replacing it by its "successor". I don't think many people would give this answer (this variant was not included in the survey). There are also four fringe answers. One of them—**qijk**—was unexpected, but actually seems to me quite reasonable and even clever: the diagonal *group* ⇒ *group*, *leftmost* ⇒ *rightmost* correspondence caused the two groups to be seen as going in opposite directions, generating a *successor* ⇒ *predecessor* slippage; thus, the leftmost letter was replaced by its *predecessor*.

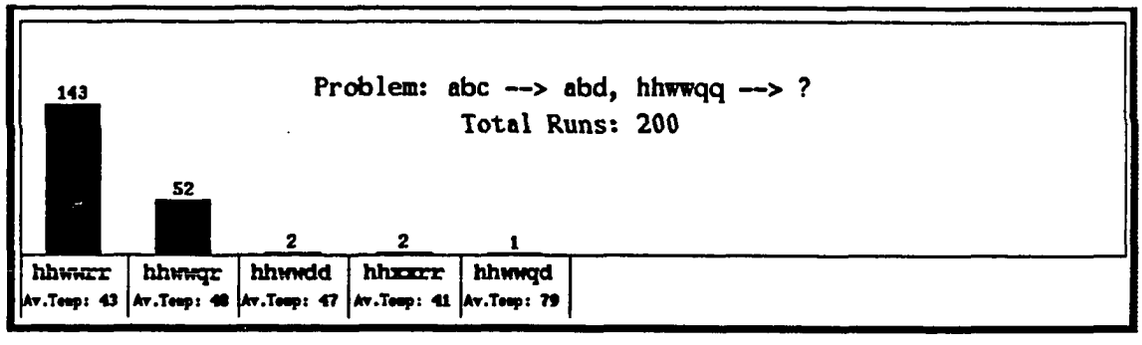
In this problem, people might describe the **abcm** ⇒ **abcn** change as something like, "Replace the only letter not in a group by its successor". This, I think, is a quite intelligent way to see the change, but Copycat is not presently able to make such a description.

5.3 Variants of "abc ⇒ abd, iijjkk ⇒ ?"

The bar graph for "**abc ⇒ abd, iijjkk ⇒ ?**"



Variant 11: **abc ⇒ abd, hhwwqq ⇒ ?**



Here there is no successor structure unifying the groups in the target string, so, unlike in "**abd ⇒ abd, iijjkk ⇒ ?**", the initial and target strings cannot be mapped on to each other as wholes. This difference is reflected in the results on this variant: Here, the ratio

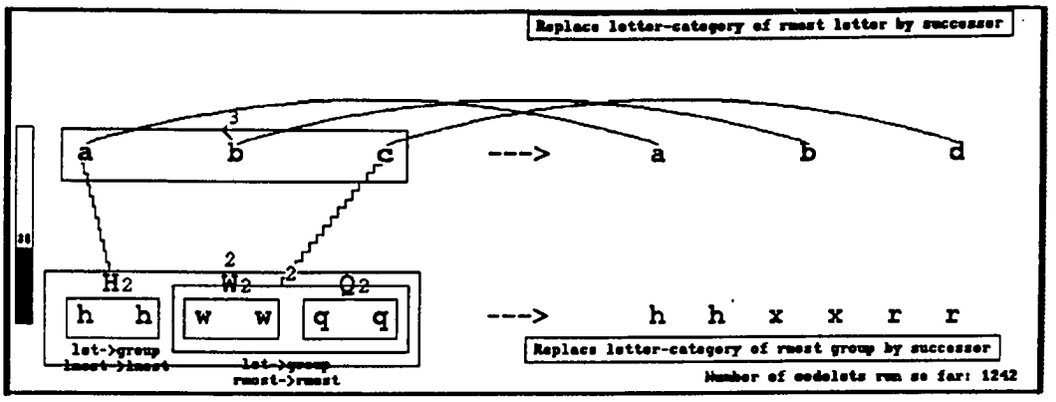


Figure 5.1: The final configuration of the Workspace on a run leading to the farfetched solution "**abc** ⇒ **abd**, **hhwwqq** ⇒ **hhxxrr**".

of "Replace rightmost *group* by successor" answers (**hhwwrr**) to "Replace rightmost *letter* by successor" answers (**hhwwqr**) is less than three to one, compared with an almost *five* to one ratio of **iijjll**'s to **iijjkl**'s in the original problem. This shows that even though the *letter* ⇒ *group* mapping is stronger than the *letter* ⇒ *letter* mapping in both problems, the whole-string mapping in the original problem serves to further support the *letter* ⇒ *group* view.

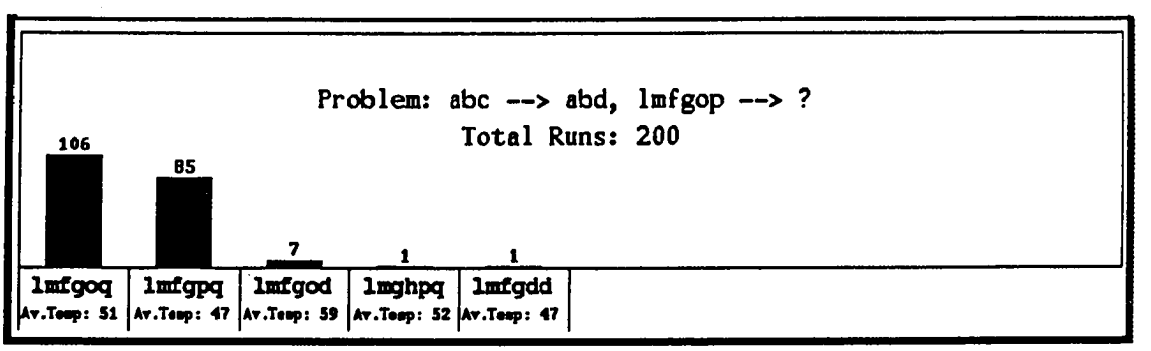
Here there are also the usual "Replace rightmost group [or letter] by *D*" answers (**hhwwdd** and **hhwwqd**), and also a ridiculously farfetched answer, **hhxxrr**, based on assigning lengths of 2 to the groups in the target string, grouping the **ww** and **qq** groups into a single group (solely on the very flimsy grounds that they have the same length), viewing that single group as the object corresponding to the rightmost letter in **abc**, and replacing it by its "successor". The final configuration of the Workspace on one of these runs is shown in Figure 5.1. Note the three levels of grouping in the target string, and the mapping of the **c** onto the "group" **wwqq**.

The fact that such an answer could be constructed two times out of 200 demonstrates some problems with program: perceiving a sameness relation between two groups of the same length (not to mention a higher-level group *based* on that sameness relation) is very strange and unhumanlike, and such behavior should be suppressed in Copycat. It would have been easy to explicitly prohibit this behavior (e.g., we could explicitly forbid sameness bonds between groups), but such an *ad hoc* prohibition is not in the spirit of this project. Rather, the prevention of such behavior should arise naturally from more general perceptual

mechanisms in Copycat. An *ad hoc* solution would only serve to cover up an interesting and unexpected way in which the program went wrong. Instead, displaying the farfetched answers Copycat occasionally gets is much more instructive and interesting for two reasons. First, these answers point out ways in which the program is lacking as a model of human perception, and second, since these answers are so unexpected, they often bring up deep issues in perception that we might not have thought of otherwise. For example, a person would never perceive **hhwwqq** in the way Copycat did in Figure 5.1. Why not? And how do people manage to avoid such bizarre ways of looking at situations? Unexpected behavior like this on the part of the program helps make it clearer just how difficult it is to understand the mental mechanisms that we are investigating.

Seventeen subjects answered this problem, and the results were not very different from those on "**abc ⇒ abd, iijjkk ⇒ ?**". The answers **hhwwrr** and **hhwwqr** were slightly closer in frequency than the corresponding answers were in the original problem (11 to 8 here versus 13 to 8 there), but there were not enough subjects to allow one to know if this difference is significant. Here, as in the original problem, people gave a number of answers that involved parsing the target string as two groups of three letters, or replacing the third letter of the string.

Variant 12: **abc ⇒ abd, lmfgop ⇒ ?**



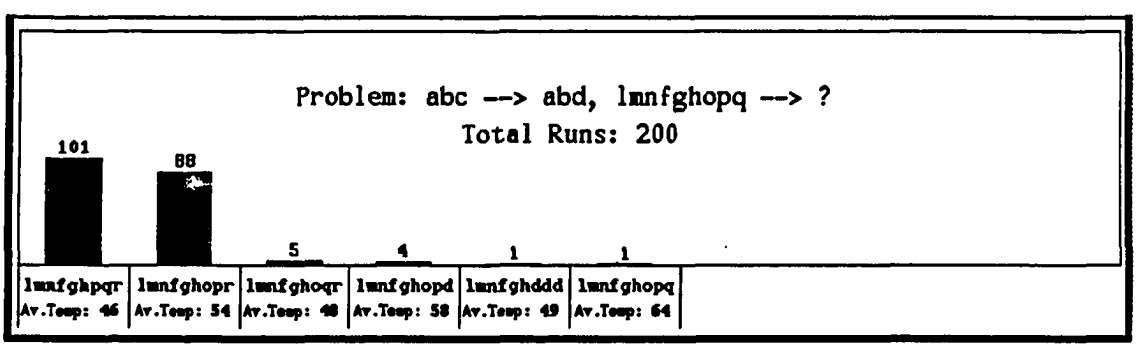
Here, we have three successor groups (or predecessor groups) rather than three sameness groups making up the target string. The former are considerably weaker than the latter, since successor and predecessor bonds are intrinsically weaker than sameness bonds.[2] Thus the program is less likely to build the three target-string groups here than it was in **iijjkk**.

[2] This intrinsic difference is meant to reflect the psychologically real difference (in the real world) between the strength of sameness bonds—as well as the speed at which they are perceived—as opposed to any other kind of bonds.

The bar graph shows that this is indeed the case: here the "Replace rightmost letter" answer (**lmfgoq**) is more frequent than the "Replace rightmost group" answer (**lmfgpq**, though the frequencies and average final temperatures are close. A more detailed statistic makes this difference even clearer: for this variant, the program constructed all three target-string groups only 49% of the time, versus 91% of the time on **iijjkk**. Here there are also some instances of "Replace rightmost letter [or group] by *D*" as well as a single instance of the farfetched **lmghpq**, which resulted from a set of events similar to those that gave rise to **hhxxrr** in the previous variant.

Twenty-one subjects answered this problem, and as was the case for Copycat, the ratio of "Replace rightmost letter" answers to "Replace rightmost group" answers was much higher than in the original problem. The answer **lmfgoq** was given 14 times, and two "Replace rightmost group by successor" answers, **lmfgpq** and **lmfgqr** (the latter of which Copycat cannot get), were given two times each. There were also three instances of answers involving the notion of "third letter in the string", and one instance of **lmfgop**, in which nothing was changed, though I am not sure what the subject's justification was.

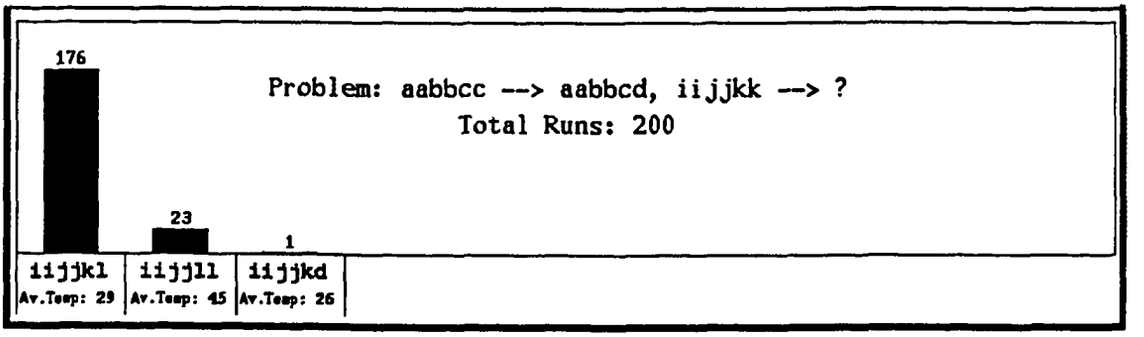
Variant 13: **abc ⇒ abd, lmnfghopq ⇒ ?**



This variant is the same as the previous one, except that the lengths of the groups in the target string are each longer by one. Since the strength of a group is a function in part of its length, it is more likely that the groups will be built here than in the previous variant. This is reflected in the bar graph: here, the "group" answer **lmnfghpqr** is more frequent than the "letter" answer **lmnfghopr**, though again, they are fairly close. Here the program constructed all three target-string groups 63% of the time, as opposed to 49% of the time in the previous variant. One of the other answers here (**lmnfghoqr**) reflects Copycat's perennial grouping problems (only the two rightmost letters of **opq** were grouped), two answers come from replacing the rightmost letter or group by **d**'s, and the answer **lmnfghopq** resulted from the rule "Replace *C* by *D*".

Ten subjects answered this problem. The "letter" answer **lmnfghopr** was given 7 times and the "group" answer **lmnfghpqr** was given only once. The answer **lmofgiopr**, in which the rightmost letter of *each* group was replaced by its successor, was given 7 times. Copycat cannot get this answer, but it is, like **lmnfghpqr**, a "group" answer, and its frequency indicates that people were perceiving the three groups here more readily than in the previous variant; there, no one gave the corresponding answer **lnfhoq**. So in this sense, this variant affected people and Copycat in a similar way: the ratio of "group" answers to "letter" answers was higher here than in the previous variant. (People also gave 4 instances of answers involving changing either the third letter of the string or the third and all following letters.)

Variant 14: **aabbcc ⇒ aabbcd, iijjkk ⇒ ?**

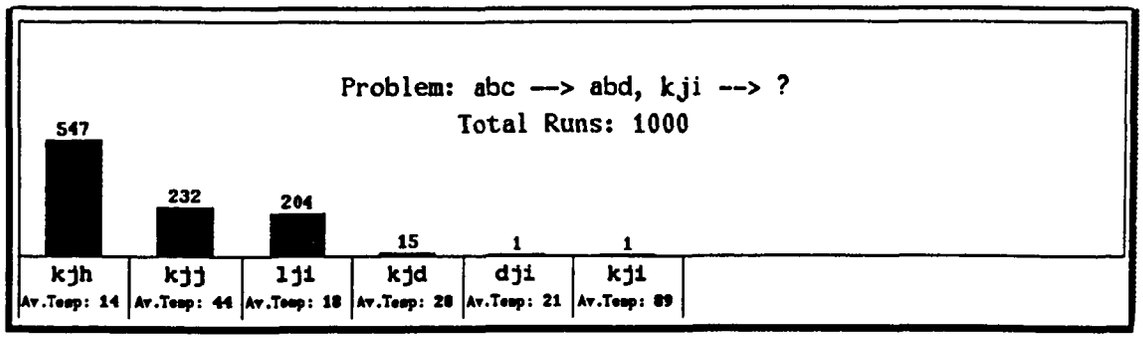


Here, the groups in **aabbcc** tend to map to the groups in **iijjkk**, and, since both strings form successor (or predecessor) groups at the group level, the two strings tend to map on to each other as wholes. All this serves to prevent the rightmost *letter* in **aabbcc** from mapping onto the rightmost *group* in **iijjkk**, which prevents answer **iijjll** from being given very often.

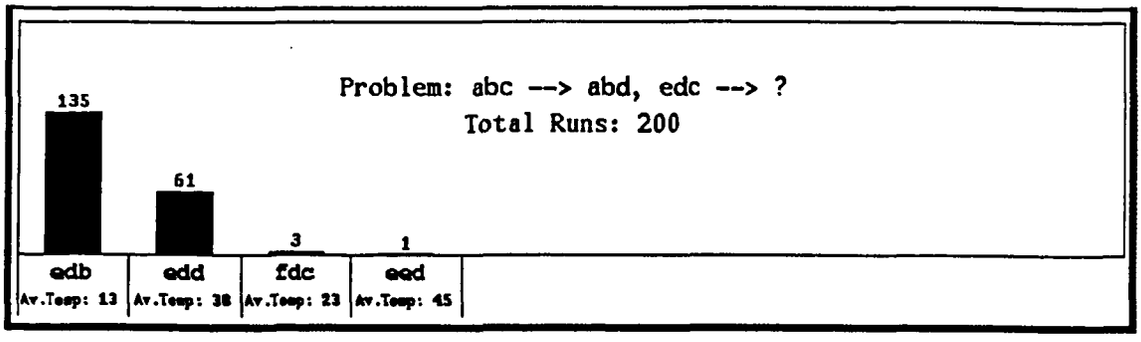
The same pressures were felt by the 10 subjects who answered this problem: all of them gave the answer **iijjkl**, and no other answers were given.

5.4 Variants of "abc ⇒ abd, kji ⇒ ?"

The bar graph for "abc ⇒ abd, kji ⇒ ?"



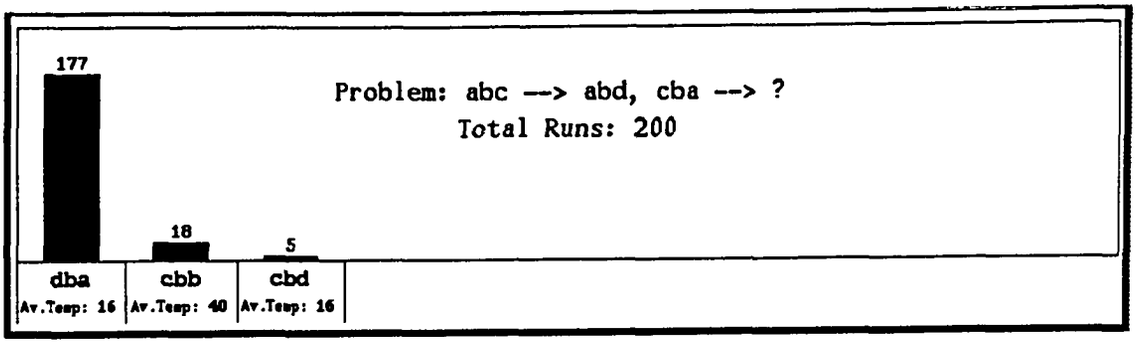
Variant 15: abc ⇒ abd, edc ⇒ ?



This variant is similar to the original, except now there is a **c** on the right, increasing the pressure to make the vertical (*rightmost* ⇒ *rightmost*) rather than diagonal (*rightmost* ⇒ *leftmost*) mapping. (This problem also fits in with variants 3–7 given above.) This pressure is reflected by the high frequencies of answers **edb** and **edd** (representing vertical mappings) as compared to **fdc** (representing the diagonal mapping). (The answer **eed** of course results from one of the bad groupings Copycat is plagued with: **e-dc.**) In fact, in this variant, vertical mappings make up 99% of the total, versus 80% in **kji**. The answer **edb** is the analog of answer **kjh** (**fdc** is the analog of **lji**), and it has the lowest temperature here.

Eighteen subjects answered this problem. Answer **edd** was given 12 times, **edb** 6 times, and **fdc** 2 times. The proportion of vertical to diagonal mappings done by people is not very different here from that on "**abc ⇒ abd, kji ⇒ ?**", where vertical-mapping answers (**kjj**, **kjh**, and **kjd**) were given a total of 12 times, and the diagonal-mapping answer **lji** was given only once. However, there weren't enough diagonal mappings made in either case (2 here, 1 there) to draw any general conclusion.

Variant 16: **abc ⇒ abd, cba ⇒ ?**

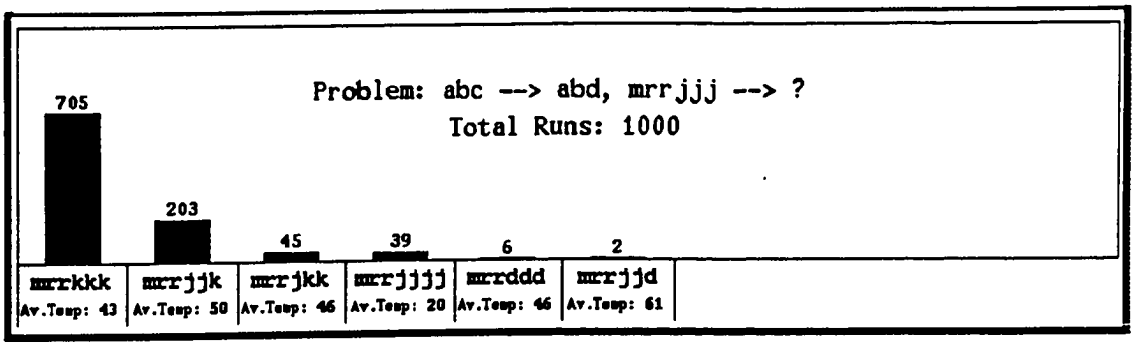


This is a variant of "**abc ⇒ abd, kji ⇒ ?**", but it also has some elements of "**abc ⇒ abd, xyz ⇒ ?**", because trying to answer the analog of **kjh** leads to a snag here (*A* has no predecessor). The bar graph shows that **dba** is by far the most frequent answer. Strong pressures lobby for this diagonal-mapping: not only are the $A \Rightarrow A$, $B \Rightarrow B$, and $C \Rightarrow C$ mappings very compelling, but also a vertical (*rightmost ⇒ rightmost*) mapping could lead to the slippage *successor ⇒ predecessor*, and then a snag. Thus **dba** by far predominates. The answer **cbb** corresponds to the answer **kjj** in the original problem, but **cbb** is much less frequent here than **kjj**, because of the strong forces described above. The answer **cbd** comes from the usual "Replace rightmost letter by *D*" rule; interestingly, every instance of it was the result of the program trying to take the predecessor of *A*, failing, and having to restructure its initial interpretation of the problem. (Its low average temperature is due to the fact that a strong whole-string mapping was made on these runs.) Even though hitting a snag is possible in this problem, the identical letter-category mappings help the program to avoid doing so most of the time. In **xyz**, the program hit the snag at least once on 98% of the runs, but here, the program made the vertical mapping (involving a *successor ⇒ predecessor* slippage) and hit the snag only 18% of the time. This is also to be contrasted with Copycat's behavior on **kji**, where the program made the vertical mapping (and got the answer **kjh**) 55% of the time.

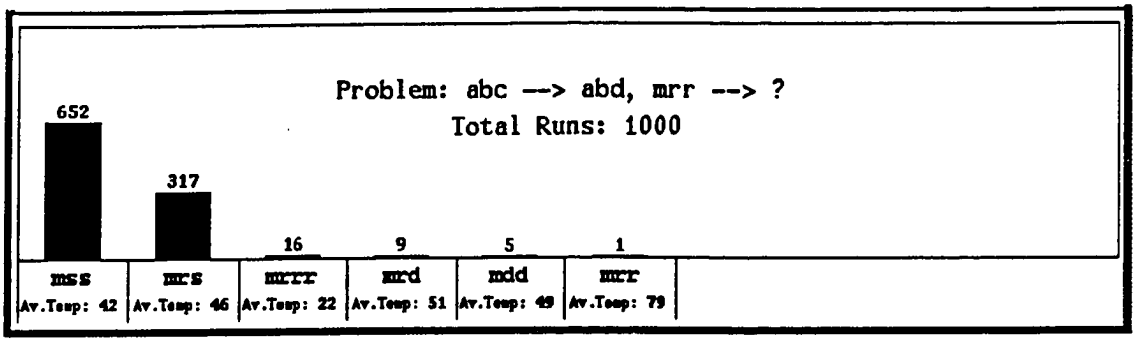
Ten subjects answered this problem. The answer **dba** was given by 7 out of 10 subjects and **cbb** was given by 6 out of 10 subjects. This is to be compared with the frequency of the diagonal-mapping answer to the original problem, **lji**, which was given by only 1 out of 10 subjects. Thus people, like Copycat, were more inclined to make the diagonal mapping here than in the original problem.

5.5 Variants of "abc ⇒ abd, mrrjjj ⇒ ?"

The bar graph for "abc ⇒ abd, mrrjjj ⇒ ?"



Variant 17: abc ⇒ abd, mrr ⇒ ?

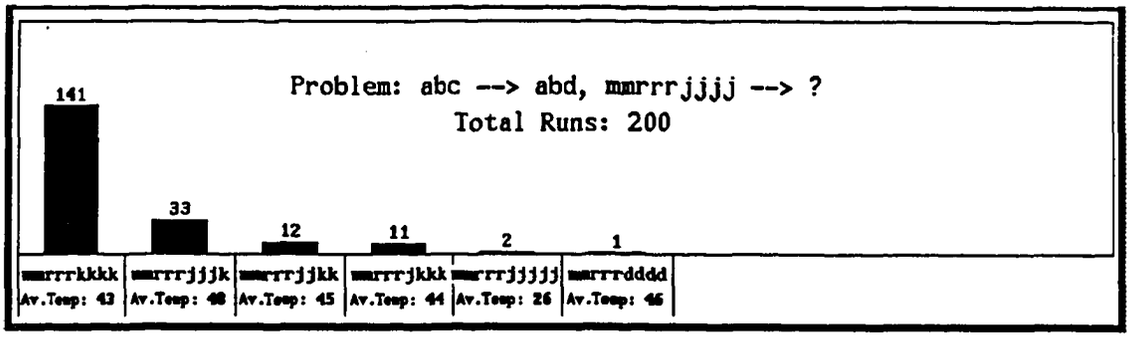


Here, the string is shortened to one m and two r's. There should be almost no pressure here for Copycat to perceive the m as a single-letter sameness group, since there is only one other possible sameness group in the target string, as opposed to two in **mrrjjj**. The answer **mrrr** (viewing **mrr** as a string whose groups increase in length, and replacing the two r's by three r's) seems to me almost completely unjustified here. The bar graph above represents 1000 runs on this problem, so it can be directly compared with the **mrrjjj** bar graph. Similar to **mrrjjj**, the top two answers by far are **mss** and **mrs**, though here Copycat answers **mrs** significantly more often than it answered **mrrjjk** in the original problem, mostly because here, given that both strings consist of three letters, there is a strong set of leftmost, middle, and rightmost letter-to-letter correspondences between **abc** and **mrr**, which wasn't possible in **mrrjjj**. This view lobbies against grouping the string as **m-rr** and having the **c** correspond to the group **rr**, which is necessary for the answer **mss**.

Copycat answered **mrrr** 16 out of 1000 times, as compared with 39 out of 1000 times for **mrrjjjj** (thus the latter occurred almost 2 1/2 times more often). But 16 out of 1000 is still too high. It would be hard to find a person who would ever give this answer seriously;

it is not justified here as **mrrjjjj** was in the original problem. (In the survey, all 10 people who were given this problem answered **mrs** and nothing else; unlike Copycat, these people did not seem to group the two **r**'s together, since no one answered **mss**.) The current version of Copycat is somewhat too willing to make single-letter groups and to perceive relations among group lengths. However, there is a dramatic difference between Copycat's behavior on this variant and on the original problem: on "**abc ⇒ abd, mrrjjj ⇒ ?**", the **m** was made into a single-letter group on 42% of the runs (only a fraction of those runs resulted in answer **mrrjjjj**), whereas in this variant, this single-letter group was made on only 4% of the runs. Since **mrr** is so much shorter than **mrrjjj**, once this single-letter group is made, it is easier in this variant than in the original problem for the program to build the other structures (length descriptions given to groups, bonds between the groups based on length relationships, etc.) that are necessary to come up with an answer in which group-length (rather than letter-category) is replaced. Of course there is more top-down support in **mrrjjj** for all these structures, but more of them to make as well. So once the single-letter group is made in **mrr**, there is a much better chance (*too* much better) that the "length" answer will be given than if the same event happens in **mrrjjj**.

Variant 18: **abc ⇒ abd, mmrrrjjjj ⇒ ?**

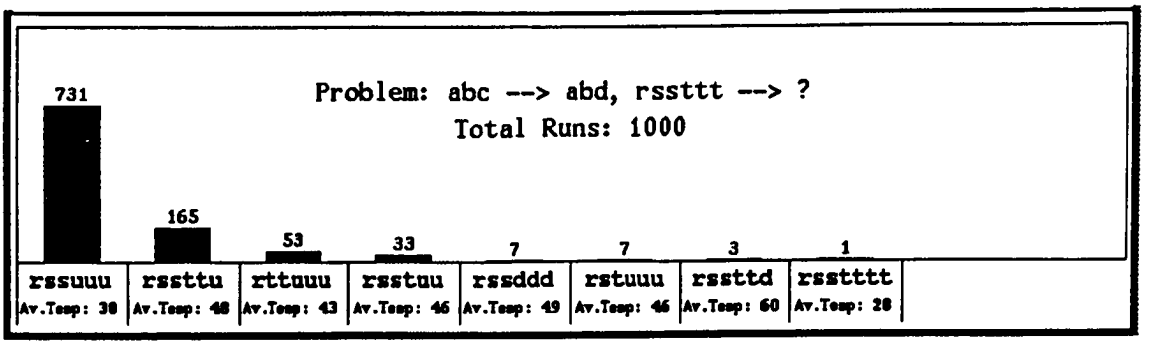


Here the target string can be parsed as 2–3–4 rather than 1–2–3. As can be seen from the bar graph, the 2–3–5 answer (**mmrrrjjjjj**) was given only two times out of 200 runs (1% of the time) as opposed to almost 4% of the time in the original problem. This is because the parsing based on group lengths is less likely to occur here than in the original problem: there, the building of a single-letter group made it more likely that group lengths would be noticed in the target string, while here, since there is no single-letter group, group lengths are noticed less often.

In the survey, 7 people answered this problem. No instances of **mmrrrjjjjj** were given. The answer **mmrrrkkkk** was given 6 times, **mmrrrjjjk** 4 times, and **mmrrrkkkkk** ("Re-

place the rightmost group by its length successor *and* its letter-category successor") 2 times. The last answer (which Copycat cannot get, since it is currently unable to build more than one kind of bond between the same two objects) shows that at least some people perceived the length-sequence here, though there were not enough subjects who gave "length" answers here or in the original problem to make a useful comparison. (There were also two instances of "Replace third letter" answers.)

Variant 19: abc ⇒ abd, rssttt ⇒ ?

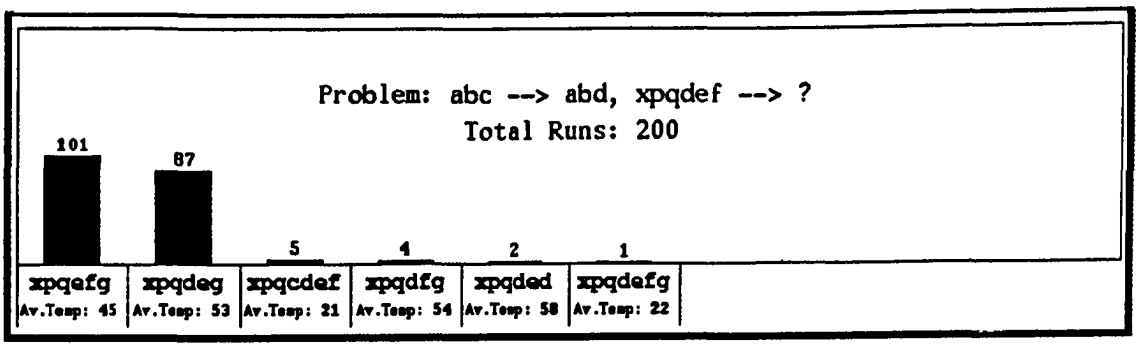


In the target string here, there are possible successor bonds both between letter categories and between group lengths, so the program should not give the length answer (**rssttt**) very often, because it is able to get a good letter-category answer (**rssuuu**); the pressure resulting from the lack of successor bonds (as in **mrrjjj**) is missing here. I ran the program 1000 times on this variant in order to show it could get **rssttt**, but it got it only once in the 1000 runs, to be compared with 39 instances of **mrrjjjj** in the original problem. In addition, the final temperature on **rssttt** here is roughly the same as the average final temperature on **rssuuu**). The other answers are similar to the answers given in the original problem (plus a few additional answers based on strange groupings of the target string).

Seven people answered this problem, and there was almost no difference between the results here and on the previous variant. Answer **rssuuu** was given 6 times, **rssttu** 3 times, and **rssuuuu** once. Again, the current version of Copycat cannot get this answer, since it is unable to build more than one kind of bond between the same two objects. The answer **rssuuuu** seems to me to be much more reasonable here than **mrrkkkk** is for the original problem. Here, since there are successor relations both between letter-categories and between group-lengths in the target string, it seems justified to give an answer that takes both types of relationships into account, whereas that justification is lacking in **mrrjjj**, where there are relationships only between group-lengths, not between letter-categories.

As for the previous variant, there are not enough subjects who give "length" answers here or in the original problem to make a useful comparison. (There were also two instances of "Replace third letter" answers.)

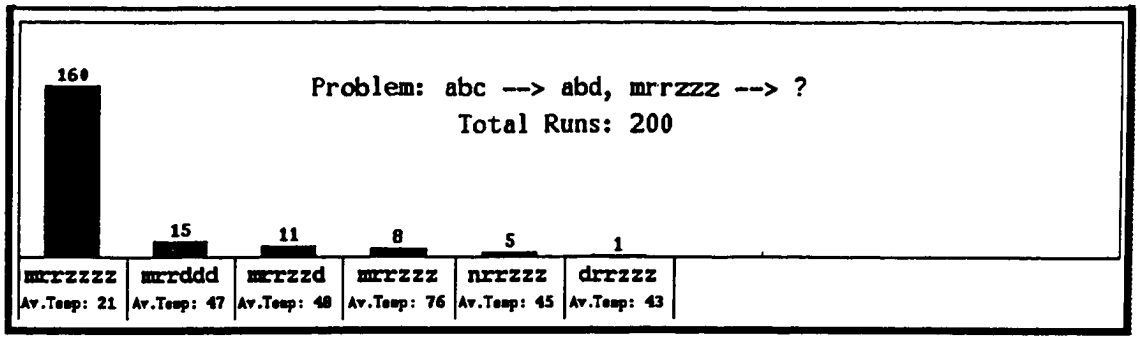
Variant 20: **abc ⇒ abd, xpqdef ⇒ ?**



Here, the target string consists of successor (or predecessor) groups (rather than sameness groups) that increase in length. As was pointed out in the discussion of Variant 12 above ("**abc ⇒ abd, lmfgop ⇒ ?**"), successor and predecessor groups are weaker than sameness groups and are not built as readily. This is reflected in the bar graph, which shows that the answer **xpqefg** ("Replace rightmost *group* by successor") is quite close in frequency to **xpqdeg** ("Replace rightmost *letter* by successor"), indicating that on a fair number of the runs the program did not build the three target-string groups. When Copycat does build the groups and notices the relations among their lengths, there are two possible answers that can be given here: if the groups are seen as right-going successor groups, then the program answers **xpqdefg**, increasing the group length to the *right*; if the groups are seen as left-going predecessor groups, then the program answers **xpqcdef**, increasing the group length to the *left*. (Most people would opt for the former, but the program does not have the same left-to-right bias that people have.) The combination of these two answers is 3% of the total versus 4% of the total for **mrrjjjj** in "**abc ⇒ abd, mrrjjj ⇒ ?**". This difference is not very significant, so even though the program perceives the target-string groups less often, it gives the length answer with roughly the same frequency. I am not sure why this is the case.

This variant was not included in the survey given to people.

Variant 21: **abc ⇒ abd, mrrzzz ⇒ ?**

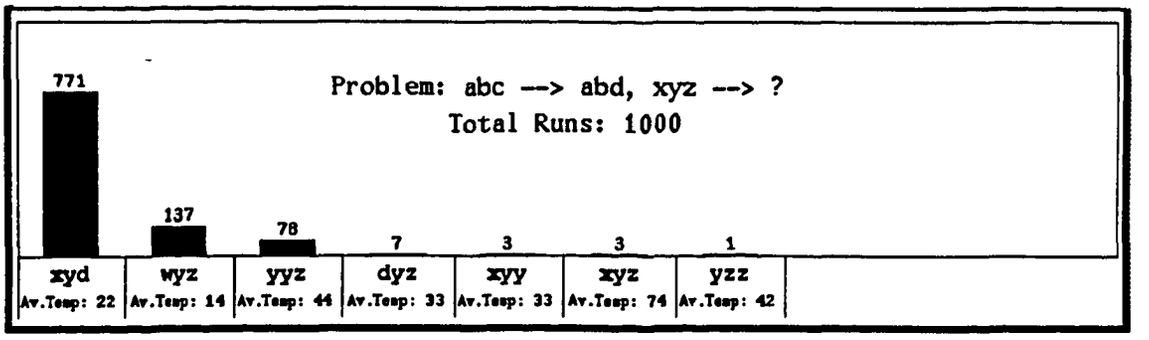


This variant combines "**abc ⇒ abd, mrrjjj ⇒ ?**" with "**abc ⇒ abd, xyz ⇒ ?**". The point of this variant was to see if the inability of the program to take the successor of *Z* would force it to notice the target-string length relations more often. As can be seen from the bar graph, this was indeed the case: the pressure of the "*Z* has no successor" snag and the resulting high temperature made the length answer (**mrrzzzz**) by far the most frequent one, comprising a whopping 80% of the total, as compared to only 4% in **mrrjjj**. Other answers include the usual ones, along with a few instances of **nrrzzz** and one instance of **drrzzz**, which were never given in the original problem. They are the analogs of **yyz** and **dyz**, and come about when a *first ⇒ last*, *rightmost ⇒ leftmost* correspondence is built. (Copycat cannot get the answer **lrrzzz** here, since there are no relationships between letter-categories in the target string, and thus there is no way for a *successor ⇒ predecessor* slippage be made.)

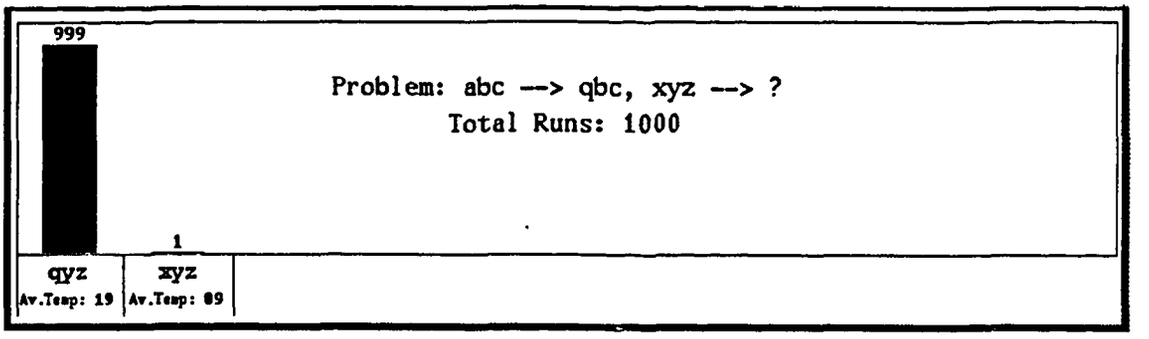
In the survey, ten people answered, and this variant did not have the effect on them that it had on Copycat. Not one instance of **mrrzzzz** was given. Instead, the subjects gave a set of answers similar to those given on "**abc ⇒ abd, xyz ⇒ ?**". All the subjects who were given this variant had immediately before been given "**abc ⇒ abd, xyz ⇒ ?**", and it is likely that they were strongly influenced by their solutions to this previous problem. It would be useful to collect more answers from people who hadn't seen the **xyz** problem first.

5.6 Variants of "abc ⇒ abd, xyz ⇒ ?"

The bar graph for "abc ⇒ abd, xyz ⇒ ?"



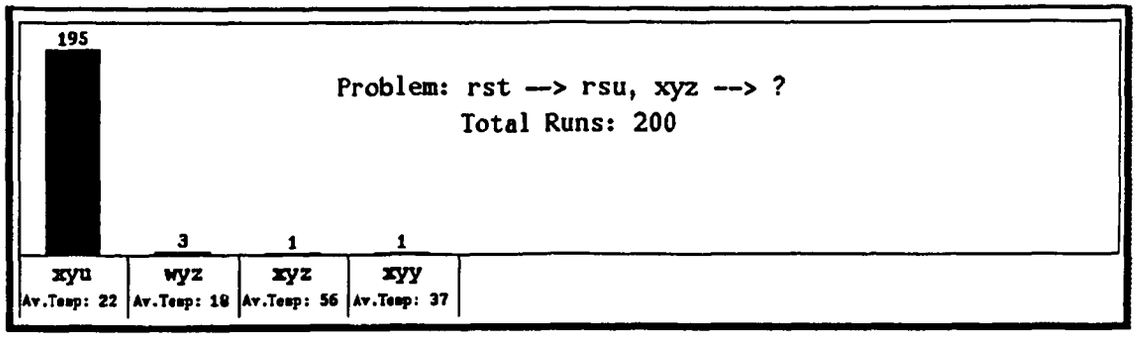
Variant 22: abc ⇒ qbc, xyz ⇒ ?



The purpose of this variant was to demonstrate that it is not easy for the program to make an **a**–**z** diagonal mapping based on *first* ⇒ *last*. As in Variant 8 ("**abc** ⇒ **qbc**, **ijk** ⇒ ?"), the possible rules here are: "Replace leftmost letter by *Q*", "Replace first letter [of the alphabet] by *Q*" (possible only if the **a** is given the description *first*), and "Replace *A* by *Q*". If either the first or second rule were constructed, and if a *first* ⇒ *last* mapping were made from the **a** to the **z**, then the answer would be **xyq**. But in 1000 runs, Copycat never made this mapping: 999 times out of 1000 it answered **qyz**, and once answered **xyz** ("Replace *A* by *Q*"). This shows it takes strong pressure to make the **a**–**z** diagonal mapping, pressure that is present in the original problem, but not here.

The results from people here were almost the same as for Variant 8: like Copycat, none of the 10 subjects given this problem answered **xyq**. Six answered **qyz**, one of those six also answered **xyz**, and the other four gave illegal answers, similar to those given on Variant 8.

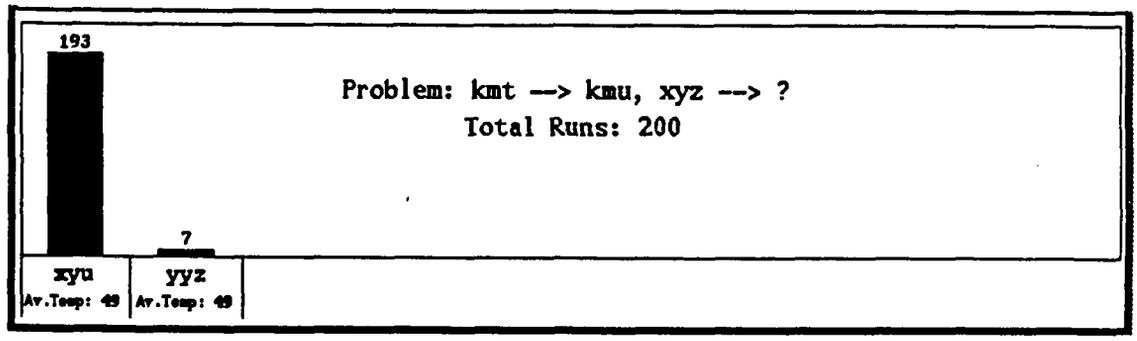
Variant 23: rst ⇒ rsu, xyz ⇒ ?



In this variant, because of the lack of an **a**, there is no *first* ⇒ *last* mapping possible to create pressure for a diagonal mapping between the initial and target strings. Accordingly, the program gave the answer **wyz** on only 1.5% of the runs, whereas this answer was given on almost 14% of the runs on "**abc** ⇒ **abd**, **xyz** ⇒ ?". In this variant, the answer **wyz** comes about solely on the basis of an unlikely *rightmost* ⇒ *leftmost* slippage (more likely here than in most problems, due to the high temperature resulting from the "*Z* has no successor" snag, but still quite unlikely). This illustrates the important role played by the *first* ⇒ *last* mapping in Copycat's **wyz** solution to the original problem.

Eleven people answered this problem, and no answers involving diagonal mappings were given, whereas on the original problem, there were several such answers given (**wyz**, **yyz**, **dyz**, and **yzz**). However, it is hard to compare the results from people here and on the original problem, since there are so few subjects here compared to the 34 subjects on the original problem. The answers here were **xyy** (4), **xy** (3), **xyz** (3), **xzz** (3), and **xz** (1).

Variant 24: kmt ⇒ kmu, xyz ⇒ ?

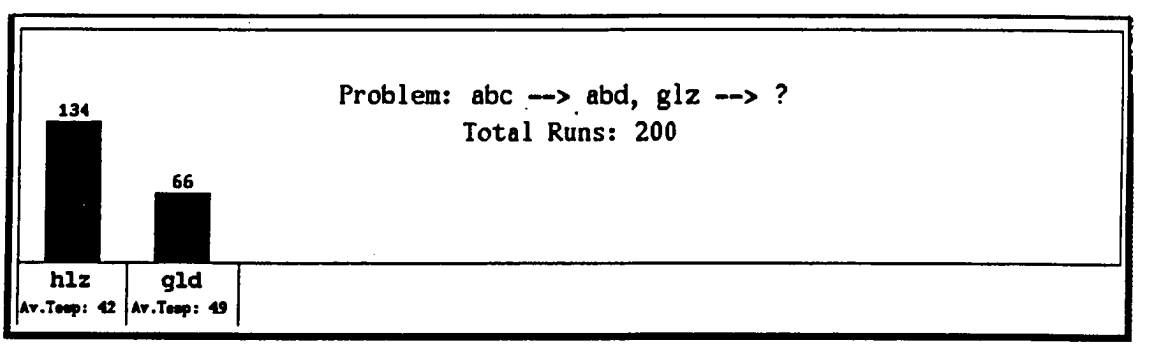


Here, not only is there no *first* ⇒ *last* mapping possible, but there are no successor relations in the initial string, so the initial and target strings cannot be mapped as wholes. The results are similar to those of the previous variant: on nearly every run the answer **xyu** ("Replace rightmost letter by *U*") is given. However, there are also some instances of **yyz**,

resulting from a fairly unlikely *rightmost* ⇒ *leftmost* mapping. Somewhat unexpectedly, this diagonal mapping is more likely here than in the previous variant (where **yyz** was not given once during the 200 runs and **wyz** was given only three times) since here there is no whole-string vertical mapping between the initial and target strings. In the previous variant, the whole-string mapping between **rst** and **xyz** supports the vertical *leftmost* ⇒ *leftmost* and *rightmost* ⇒ *rightmost* correspondences, which compete with diagonal correspondences. Here, the vertical correspondences have less support, so the diagonal mapping has more of a chance. But since no whole-string mapping can be made here, the answers **wyz** and **xyy** are not possible. In the original problem and in the previous variant, these answers result from a *successor* ⇒ *predecessor* slippage when the whole initial and target strings are mapped onto each other in opposite alphabetic directions.

This variant was not included in the survey given to people.

Variant 25: **abc ⇒ abd, glz ⇒ ?**

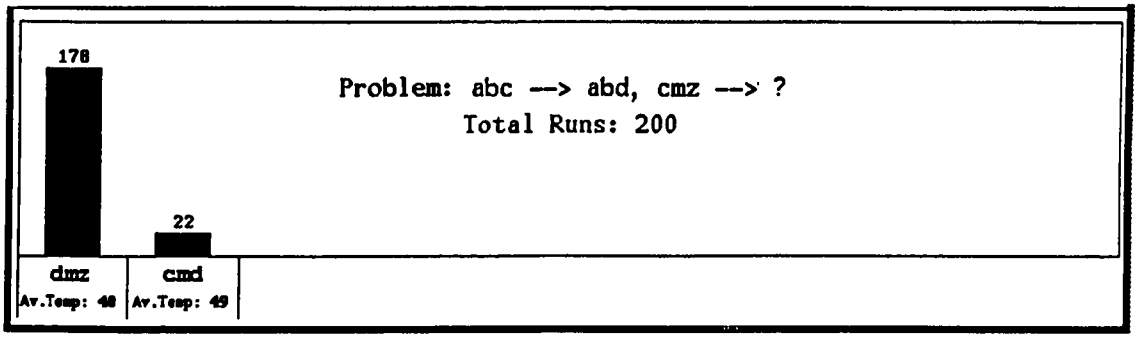


Here there *is* a possible *first* ⇒ *last* mapping between the **a** and the **z**, but since the target string is not a successor group, there is no possible whole-string mapping between it and the initial string. Thus, as in the previous variant, if the **a** and **z** are seen to correspond here, and the *leftmost* letter is changed, it can only be changed to its successor. Here the frequency of the "Replace *leftmost* letter by successor" answer (**hlz**) exceeds that of the "Replace rightmost letter by *D*" answer (**gld**) by a considerable amount, a dramatic difference between the results here and on the previous variant, as well as between these results and those on the original problem. The reason **hlz** predominates here is that the *first* ⇒ *last* mapping makes the diagonal **a**–**z** correspondence strong, and, in contrast to the original problem, this correspondence doesn't face much competition from vertical correspondences, since no whole-string same-direction mapping supports them. So in this case, given the high temperature due to the *Z*-snag and the intrinsic weakness of the "Replace rightmost

letter by *D"* rule, the *first* ⇒ *last* correspondence gets built more easily, and does not face strong competition from incompatible bonds, groups, and correspondences.

This variant was not included in the survey given to people.

Variant 26: **abc ⇒ abd, cmz ⇒ ?**



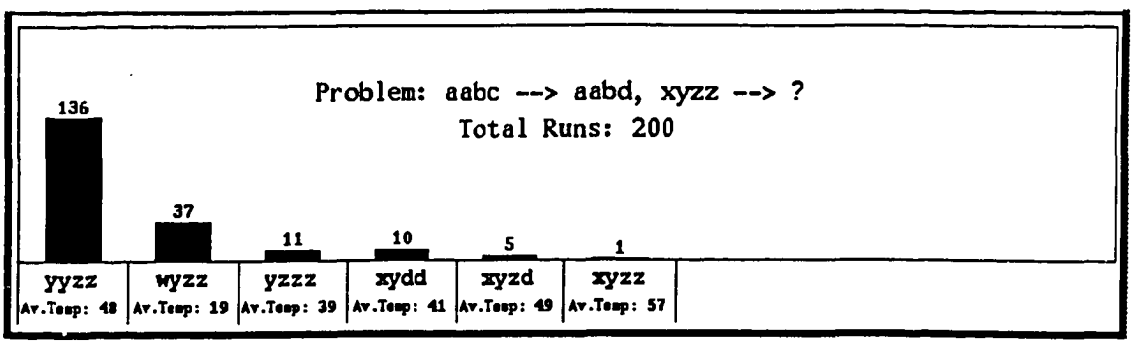
This variant gives Copycat the possibility of an easy out from the impasse: if it can't take the successor of *Z*, it can resort to changing the **c** in the target string instead. As can be seen in the bar graph, on most of its runs, the program either took advantage of this escape route or avoided the snag altogether, answering **dmz** 89% of the time. This can be compared with the previous variant ("**abc ⇒ abd, glz ⇒ ?**"—which has no **c** in the target string), in which a leftmost-letter-change answer was given 67% of the time, and with the original problem, in which answers involving a leftmost-letter change were given only 22% of the time. It is also interesting to compare this with Variant 7 ("**abc ⇒ abd, cmg ⇒ ?**") which is the same as this variant except that there, the rightmost letter was an instance of *G*, which has a successor. There, a leftmost-letter-change answer was given only 31% of the time. So here, given the snag and the resulting high temperature, mapping the **c**'s becomes more compelling, even though a *rightmost* ⇒ *leftmost* slippage has to be made.

As would be expected from the results on Variant 7, the presence of the **c** here allowed the program to sidestep the snag entirely on 30% of the runs and go straight to the answer **dmz**, whereas on the original problem, Copycat sidesteps the snag and goes straight to a leftmost-letter-change answer on only 0.3% of the runs.

Ten people answered this problem, but only one answered **dmz**, which is consistent with the answers people gave on "**abc ⇒ abd, cmg ⇒ ?**" (Variant 7), where the two **c**'s were also unlikely to be mapped. It seems that most people are much less likely than Copycat to notice superficial similarities such as that between the two **c**'s, even when other ways of making an analogy fail. However, all the subjects who solved this problem had first solved "**abc ⇒ abd, xyz ⇒ ?**" and Variant 21 ("**abc ⇒ abd, mrrzzz ⇒ ?**"), and it may be that

they were strongly influenced by their solutions to those previous problems (the answers given in the survey here were similar to the set given for Variant 21). Again, it would be useful to collect more answers from people who hadn't seen these other problems first.

Variant 27: **aabc ⇒ aabd, xyzz ⇒ ?**



Here, the pressure to make a diagonal mapping is increased because of the *sameness-group* ⇒ *sameness-group* mapping between the **A** and **Z** groups. The results show that this new pressure makes a big difference: answers involving a *leftmost* ⇒ *rightmost* mapping (here **yyzz**, **wyzz**, and **yzzz**) make up 92% of the total, versus only 22% of the total in "**abc ⇒ abd, xyz ⇒ ?**". It is also interesting to compare this variant with the original in seeing how much the *group* ⇒ *group* mapping helped Copycat to avoid trying to take the successor of *Z* and failing (the "snag"). In "**abc ⇒ abd, xyz ⇒ ?**", as was noted earlier, this snag is very rarely avoided: in 1000 runs, only 2% of the answers were gotten without first running into the snag (most of these runs resulted in answer **xyd**), and there were an average of 9 snags per answer (as mentioned earlier, the program got into the snag on average 9 times before getting an answer—this loopish behavior will be discussed further in the next chapter). In this variant, Copycat avoided the snag on 20% of the runs, and the average number of snags per run was 4. So the presence of the two sameness groups helps the program considerably to avoid or get out of the snag.

It is also interesting to compare this variant with Variant 9 ("**aabc ⇒ aabd, ijkk ⇒ ?**"). It should be expected that Copycat would get answers based on a diagonal (*group ⇒ group, rightmost ⇒ leftmost*) mapping here more often than it did in Variant 9, since here the vertical mapping leads to a snag. Indeed, diagonal-mapping answers make up 92% of the total here versus only 47% of the total in Variant 9.

Ten people answered this problem, and it seems that, as was the case for Variant 9, they weren't affected as strongly by the *group ⇒ group* mapping as Copycat was. There were 2 instances of diagonal-mapping answers (**yyzz** and **wyzz**), but again it is hard to compare

the results from people here and on the original problem, since there were so few subjects here compared to subjects on the original problem. The other answers people gave here were **xyzz** (2), **xyzy** (2), **xyyy** (1), **xzzz** (1), **xyzd** (1), **xydd** (1), **xyz** (1), and **xy** (1).

5.7 Summary

The 27 variants in this chapter demonstrate the range of Copycat's abilities, and how different constellations of pressures affect its behavior. Of course, many more variants could have been included, but the ones given are enough to give the reader a good sense of the program's behavior and "personality".

It may seem to the reader that what Copycat does on many of these variants is simply what one obviously *should* do, given the pressures that are present. But this is precisely the best argument for the model's plausibility: it is flexible enough to to adapt to all these different situations and to act in appropriate ways. Copycat also is able in some cases to make analogies that are not at all obvious, and that demonstrate a fair degree of insight.

The other side of the coin, of course, is represented by the bad analogies that the program makes, which reveal its internal flaws and weaknesses. But they also demonstrate that Copycat has the *potential* to get farfetched answers—a potential that is essential for flexibility—and yet manages to *avoid* them almost all the time, which demonstrates its robustness.

Copycat's performance on the variants to the original five problems demonstrates the program's robustness and flexibility as it is "stretched"; it shows how well the program continues to perform as it is pulled away, little by little, from the most central problems that it was deliberately designed to solve. The program was not designed to work specifically on these variants; in fact, in almost every case, the program was not tested on the variants until after it had been completed. Copycat's performance on these variants thus gives evidence for the generality of the mechanisms that we are proposing and modeling.

This chapter, in demonstrating the range of Copycat's intelligence (as well as the ways in which it lacks intelligence), has expanded on Chapter 4 in addressing the AI criteria for judging this project. I would argue as well that these demonstrations of the program's flexibility also address to some extent the psychological criteria, since the extent to which the program performs with flexibility over a range of different situations and demonstrates its ability to deal with general issues in perception and analogy-making—the extent to which it demonstrates that it has human-like *concepts*—lends plausibility to it as a model

of some central aspects of human intelligence.

5.7.1 Summary of the Comparisons With People

The results of the survey given to people serve two purposes: they show how well Copycat matches people in the range of answers it gets to various problems, and also to what extent Copycat and people are similarly affected by variations in pressures.

Comparisons With People on Range of Answers

The survey produced a fairly comprehensive list of the answers given by people to many of these letter-string problems (although on other problems there were too few subjects to get a complete range). Copycat is able to get a large number of these answers: it can get about half of the answers people give overall, and it can get almost all of the answers given by three or more people. Most of the answers Copycat misses fall into three main classes:

1. *Answers involving descriptions of the numerical position of letters in the target string.* (E.g., "third letter", "leftmost two letters".) For example, some people gave answers such as "**abc ⇒ abd, iijjkk ⇒ iikjkk**" ("Replace third letter by successor") or "**abc ⇒ abd, iijjkk ⇒ iikkll**" ("Replace all letters after the leftmost two by their successors"). Copycat is currently unable to make such descriptions; it does not have concepts such as "third" or "leftmost two".

2. *Answers involving groupings not based on bonds between letters.* For example, several people answered "**abc ⇒ abd, mrrjjj ⇒ mrrjkk**", parsing the target string as **mr-rj-jj** based on pressure to see three equal-length elements, as in **abc**. Copycat is currently unable to group letters unless there is a bond between them.

3. *Answers involving descriptions of letters with respect to groups.* For example, several people gave the answer "**abc ⇒ abd, lmnfghopq ⇒ lmofgiopr**", using the rule "Replace the rightmost letter of each successor group by its successor". Copycat is currently unable to make descriptions such as "rightmost letter of successor group".

All of these discrepancies point to abilities that Copycat lacks. Giving Copycat these abilities would involve extending the description-making and grouping mechanisms that the program already has. Making these extensions would be a worthwhile direction to take in future work on this project.

There were also other answers given by people that Copycat is unable to get, but they are harder to classify. In particular, some of the answers given by people to the problem "**abc ⇒ abd, xyz ⇒ ?**" involve concepts and intelligence far beyond Copycat's. For example, a very common answer is **xy**, for which people use imagery such as "the **z** falls off the edge of the alphabet", making an analogy between the edges of the linear alphabet and the edges of a cliff off of which things can fall. Copycat, of course, has no such imagery (it has no imagery at all, unless knowledge such as "letter sequences are similar to number sequences", or "left-going is similar to right-going" could be be counted as a primitive form of imagery). People also sometimes answer **xzz**, reasoning that if you can't change the *rightmost* letter, then the next best thing is to change the *next-to-rightmost* letter, or **xyy**, reasoning that if you can't take the *successor* of the rightmost letter, then the next best thing is to replace it by its *predecessor*. These slippages do not come from correspondences with anything in **abc**; such slippages are made only because the analogy-maker cannot do the desired thing and thus does something close to it. Copycat currently cannot make such slippages, though I believe that this is a very important ability for general intelligence, and giving such an ability to Copycat would make it a much more flexible program. This, again, is a topic for future research. A third answer people occasionally give (jestingly) is **abd**—that is, "Replace the entire string, whatever it is, by **abd**". Even though this answer is given only in jest, the fact that it is given at all shows that people are able to describe the **abc ⇒ abd** change in that way. Copycat ideally should be able to come up with such a rule in principle, though in practice its construction should be extremely unlikely.

People also gave answers that demonstrated more flexible views of the notion of successorship than Copycat has. For example, one person in the survey answered "**abc ⇒ abd, lmfgop ⇒ lmfgqr**", seeing **qr** as the "successor" of the group **op**. Copycat currently can only give **pq** as the successor of **op**. Also, one person in the survey (and a number of people in more informal surveys) answered "**abc ⇒ abd, rssttt ⇒ rssuuuu**", replacing the rightmost group by both its alphabetical *and* numerical successor. Again, this seems to me to be a very good answer to this problem since **rssttt** has both alphabetical and numerical successorship relations, but Copycat is currently able to construct only one bond between two given objects in a string (e.g., it cannot build both length and letter-category successor bonds simultaneously). Extending Copycat's bond-building capabilities in this way is another topic for future research.

Copycat also gives a number of answers that people never give. These fall into three main classes:

1. *Bad-grouping answers*, such as "**abc ⇒ abd, iijjkk ⇒ iijkll**".

2. *Answers involving unmotivated slippages*, such as "**abc ⇒ abd, ijk ⇒ ijj**", which was based on a view in which a correspondence involving the slippage *successor* ⇒ *predecessor* was made without sufficient reason.

3. *Answers based on unmotivated uses of group lengths.* These include "**abc ⇒ abd, hhwwqq ⇒ hhxxrr**" (Variant 11) as well as "**abc ⇒ abd, cab ⇒ cabc**" (Variant 6) and "**abc ⇒ abd, mrr ⇒ mrrr**" (Variant 17). All these answers were discussed earlier in this chapter.

These are the classes of unrealistic answers that came out of Copycat's performance on the letter-string problems discussed in this and the previous chapter. If more problems were added, other such classes would likely become apparent. The answers that Copycat gets but that people never get illustrate certain problems with the model (some of these will be discussed in the next chapter). It is encouraging, though, that these are always fringe answers produced very rarely by the program, showing that even though it has the capability to produce them, it avoids them almost all of the time.

I did not include "frame blend" answers, such as "**abc ⇒ abd, xyz ⇒ dyz**", in the three classes given above. It is true that no one in the survey gave this particular answer, but people did give answers that involved similar (though perhaps less farfetched) kinds of frame blends. Moreover, people have proposed **dyz** and other such answers in jest, which means that they do actually come to mind. Thus it is *desirable* that Copycat have the ability to get such answers, though, as with the other fringe answers, it is also desirable that it not get them very often.

Comparisons With People on Effects of Variations in Pressures

The point here was, again, not to compare the frequencies of various answers people gave with the frequencies of various answers given by Copycat, but rather to see if variations in pressures caused similar *shifts* in frequency of the *types of answers* given by people and by Copycat. For example, the people in the survey and Copycat both were more likely to give answers involving a $C \Rightarrow C$ correspondence in Variant 6 than in Variant 5.

The overall results here were mixed. Of the 27 variants, 23 were included in the survey of people. On 11 out of the 22 problems, people seemed to feel the effects of the variations in pressures similarly to Copycat (these were variants 1, 2, 3, 4, 6, 8, 12, 13, 14, 16, and 22). On 7 out of 22, there seemed to be significant differences between the effects on people and on Copycat (variants 5, 7, 9, 17, 21, 26, and 27), and for the other 5 (variants 11, 15, 18, 19, and 23), there were not enough subjects giving a particular type of answer (e.g., a "diagonal-mapping" answer or an answer that involved relations between group-lengths) in order to compare with subjects giving that same type of answer in the original problem.

The main differences between the effects on people and on Copycat were:

- People were less likely than Copycat is to make mappings between two objects on the basis of their letter-categories. For example, on Variant 5 ("**abc ⇒ abd, cde ⇒ ?**"), people were much less likely than Copycat to map the two **c**'s.

- People were less likely than Copycat to perceive or make mappings between groups. This difference was clear in the responses to Variant 17 ("**abc ⇒ abd, mrr ⇒ ?**"): in the survey, no one gave the grouping answer **mss**. This difference was also seen in the responses to Variant 9 ("**aabc ⇒ aabd, ijkk ⇒ ?**") and to Variant 27 ("**aabc ⇒ aabd, xyzz ⇒ ?**"), in which very few people made the *group ⇒ group, leftmost ⇒ rightmost* mapping, thus changing the *leftmost* rather than *rightmost* letter of the target string. Copycat was more likely than people to make this mapping. However, it may be that many people would have preferred this mapping if they had seen it. People often find an answer compelling once it is pointed out to them, even if they themselves did not think of it (for instance, this is the case for many people with the answers "**abc ⇒ abd, mrrjjj ⇒ mrrjjjj**" and "**abc ⇒ abd, xyz ⇒ wyz**"). The results of a survey asking people to *rate* different *given* answers to the five target problems is given in Appendix D; it would be useful to include the variants discussed in this chapter in future such surveys of people.

- People were less likely than Copycat is to notice successorship among group lengths, even when they were faced with an impasse, as in Variant 21 ("**abc ⇒ abd, mrrzzz ⇒ ?**"). Again, it may be that once this property was pointed out to people, many would find it compelling, but it seems not to come to mind for most people when they are solving such problems themselves.

As would be true for any restricted domain, people's answers here might be influenced by

a large number of factors (e.g., previous problems they solved, extraneous knowledge about letters and letter-strings, assumptions about what they are "supposed" to answer) that do not influence the program, and for this reason there is some difficulty in interpreting the results of these comparisons. Also, Copycat has its own biases, and is not meant to match the *average* behavior of a population of people; rather, it is meant to model something more akin to a single individual, who has their own individual biases. For example, Copycat may be more inclined than *most* people to notice groups in a string, but this is not a bad thing if this bias is not an *implausible* human one. Also, Copycat may be more persistent and patient than most people in exploring possible ways of solving these letter-string problems; most people tend to give up after a short time, without thinking very hard about the problem. Again, Copycat's behavior is not implausible for a person, though it might not match that of the majority of people. Ideally, Copycat is meant to produce not just *reasonable* behavior, but also *insightful* behavior, and it thus it *should* get answers (e.g., "**abc** ⇒ **abd**, **mrrjjj** ⇒ **mrrjjjj**") that very few people come up with, but that many people at least *recognize* as reflecting an insightful and flexible use of concepts.

These comparisons have given some evidence for the program's plausibility, to the extent that Copycat has matched the range of people's answers as well as the effects of variations in pressures on people. The comparisons have also pointed out some flaws of the program and indicated some directions for future work on the project. However, the main criteria for judging the success of the program should be those given at the end of Chapter 2: Does the program exhibit flexible and insightful behavior in its microworld? Does it act, at least to some degree like it has *fluid* concepts, as people do? Does it help us to better understand what concepts are?

CHAPTER VI

SOME PROBLEMS WITH THE MODEL

The bar graphs and screen dumps in the previous two chapters have demonstrated most of the mechanisms in Copycat, and in doing so have showed off not only the program's strengths, but many of its weaknesses as well. In this chapter I will discuss some of these weaknesses, and their general implications for models of high-level perception. The point of the chapter is not to detail wholly new abilities the program would need in order to solve a wider range of problems (e.g., the ability to construct more complex rules, the ability to build bonds between non-adjacent objects in a string, or the ability to form new concepts, such as "double successor", from existing Slipnet nodes), but rather, to discuss some problems with the mechanisms the program currently has. (A discussion of possible extensions to Copycat will be given in Chapter 9). For the purposes of this chapter I will discuss two of the more salient and serious problems of the program: problems concerning top-down forces and focus of attention, and problems concerning self-watching.

6.1 Problems with Top-Down Forces and Focus of Attention

Top-down (expectation-driven) forces are an essential part of perception in general. This point is brought home very clearly by looking at some of the difficulties Copycat has in solving analogy problems in its microworld. One of the major weaknesses of the program as it now stands is that top-down pressures in the system are often not strong enough. This can be seen in Copycat's performance on problems involving long strings. For example, consider the problem "**abc ⇒ abd, ijklmnop ⇒ ?**" (Variant 1 from the previous chapter). Here, once the notion of successorship (or equivalently, predecessorship) is deemed to be highly relevant, top-down forces should take over almost completely and very quickly build

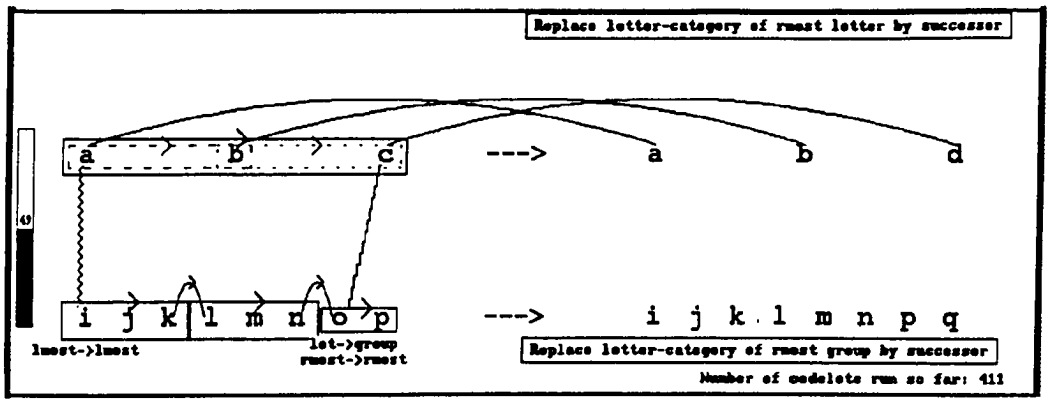


Figure 6.1: A "bad-grouping" answer.

successor bonds throughout the target string. As has been demonstrated, such forces exist in Copycat, but at present they are not strong enough. Sometimes successor bonds are built too slowly, and groups are formed out of chains of bonds that cover only part of the string. Figure 6.1, which shows the final configuration of the Workspace from a run on this problem, illustrates a case where this happened.

As can be seen in the figure, the target string has been divided into three separate successor groups (**ijk-lmn-op**) instead of one successor group comprising the whole string. What happened here was that successor bonds along the edges of the string were built fairly quickly, and then these bonds were grouped, leaving the middle letters out. Only later were bonds built in the middle, and a separate middle group was built out of them. In many cases, a single whole-string group will successfully compete against smaller groups such as these, but in this case that didn't happen, and the program answered **ijklmnpq**, replacing the rightmost group of two by its "successor".

The results of similar bad groupings were seen in some of the bar graphs presented in the previous two chapters. For example, on the problem "**abc ⇒ abd, iijjkk ⇒ ?**", the program came up with several bad-grouping answers. Screen dumps showing the final Workspace configuration for two of these answers are given in Figures 6.2 and 6.3.

In Figure 6.2, the program never grouped the two **j**'s, and instead built successor bonds from the group **I** to the leftmost **j** and from the rightmost **j** to the group **K**. This led to a parsing of the target string as two higher-level groups: **I-j** and **j-K**. The **c** was then mapped onto the rightmost of these two groups, all of whose letters were replaced by their successors in the answer.

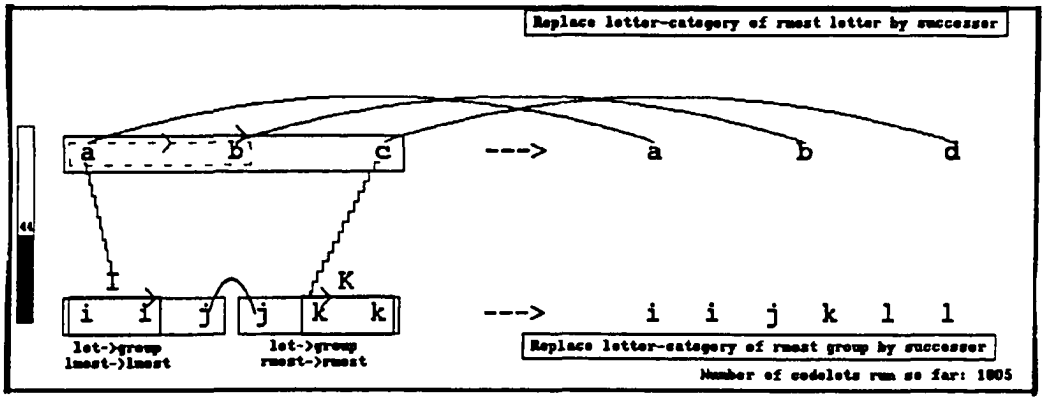


Figure 6.2: Another bad-grouping answer.

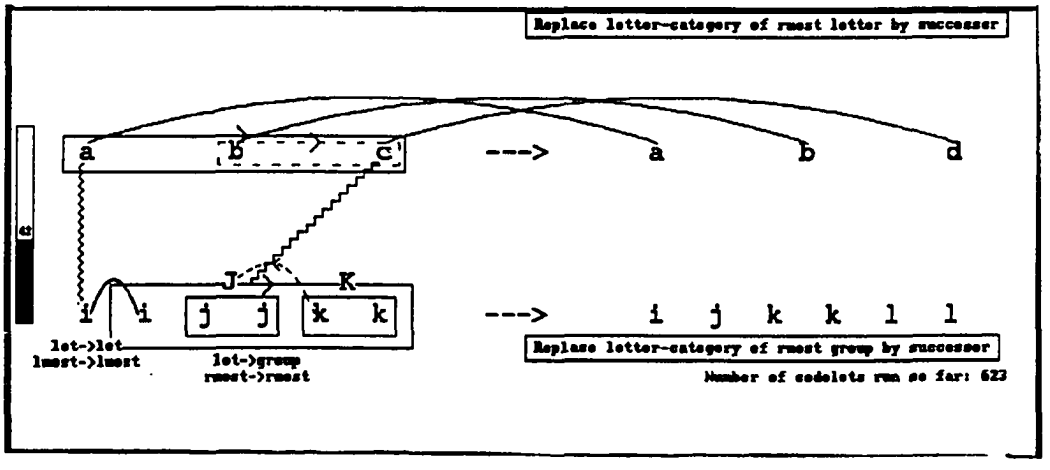


Figure 6.3: A third bad-grouping answer.

In Figure 6.3, the program never grouped the two **i**'s, and instead parsed the string as **i—i-J-K**. Thus at the top level the string was seen as consisting of two elements, the leftmost letter **i** and the group **iJK**. The **c** was seen to correspond to the rightmost element, the group **iJK**, all of whose letters were replaced by their successors in the answer. This answer, even more than the previous two, is extremely farfetched and unhumanlike.

One problem seems to be that Copycat's top-down codelets are too global: they are not targeted specifically enough. A top-down successor-bond codelet will attempt to build a successor bond *anywhere*, but it seems that what is needed here is codelets that try to build specific types of bonds in *specific* places. For example, in the problem above, once a sameness bond has been built between the two **j**'s, there should be top-down pressure to try to build the same type of bond in the adjacent position, between the **i**'s. There should be similar pressure in the problem "**abc ⇒ abd, ijklmnop ⇒ ?**": once successor bonds have begun to be built, top-down forces should try to build successor bonds adjacent to the already-existing ones.

A mechanism for implementing these kinds of specific top-down pressures would enable the program to follow what might be a more plausible route to the solution "**abc ⇒ abd, mrrjjj ⇒ mrrjjjj**". If the **R** and **J** groups have been given length descriptions and a successor bond has been created between their lengths, then there should be top-down pressure to build the same type of bond in an adjacent position—namely, between the **m** and the **R** group. This specific goal would create pressure to perceive the **m** as a group of length 1, so that a successor bond could be built with the group of length 2. As was seen in some of the variants in the previous chapter, the program is currently somewhat too willing to build single-letter groups without sufficient pressure. It seems more plausible that constructing such an unusual group should be done in response to a strong *location-specific* top-down pressure like the one described above, rather than (as is currently the case) in response to more general pressure from other groups in the string.

In general, Copycat needs a mechanism for conceiving of a specific desired structure (e.g., a successor bond between the group lengths of specific objects) in response to top-down pressure, and trying to build the necessary prior structures (e.g., a single-letter group) whose existence would make it possible to build the desired specific structure.

Part of Copycat's problem with interpreting long strings such as **ijklmnop** has to do with the program's focus of attention. One problem is that not enough attention (in the form of codelets) is directed to parts of the string that need attention (i.e., unhappy objects,

such as the middle letters of **ijklmnop** early on in the run when no bonds involving them have been built). In the current version of Copycat, codelets are indeed biased to choose unhappy objects to work on (since salience depends in part on unhappiness), but it seems that this mechanism is not working well enough to avoid occasional bad groupings like the example shown here.

Finally, Copycat has no mechanisms for either *extending* already existing groups (though a large group can fight against smaller subgroups inside it) or *merging* two adjacent disjoint groups. The ability to fluidly extend, merge, split, and, more generally, change the boundaries of groups seems very important in high-level perception (Hofstadter, 1983), and such mechanisms would certainly help Copycat in cases like the ones shown above.

The answers displayed in the screen dumps above (and other strange answers) are Copycat's "misspun tales", corresponding to the strange and humorously nonsensical stories occasionally generated by the program Talespin (Meehan, 1976), which contrasted with the more coherent, meaningful stories that it was meant to generate. As was the case for Talespin, Copycat's misspun tales are often windows onto the program's internal deficiencies, such as those discussed in this chapter. It must be said, however, that not all instances of strange answers should be considered evidence for problems with the program. As has been pointed out, the *potential* availability of *all* paths of exploration is essential for the program's flexibility (and the current program's flexibility is limited by the fact that it cannot follow all the possible paths that people could follow). Contrary to what one might initially suppose, it should be considered positive evidence for the program's strength that strange answers (such as those above, or frame-blend answers such as "**abc ⇒ abd, xyz ⇒ dyz**") do appear *occasionally*, since their existence proves that the program is indeed *capable* of following bad pathways, and yet manages to steer clear of them almost all the time. This is the kind of behavior that we want to see in Copycat. On the other hand, bad-grouping answers (and some other types of bad answers) tend to show up *too often* in the current version of the program, which indicates problems (of the kind discussed above) with the way the program is working right now.

6.2 Problems with Self-Watching

An absolutely essential feature of conscious cognition, whose necessity is shown quite clearly by some of Copycat's weaknesses, is *self-watching* (sometimes called "meta-cognition"): an ability to perceive patterns in one's own mental activities. In Copycat, temperature acts as

a primitive self-watching mechanism, in which information about the state of the program's progress toward an answer feeds back into the program's behavior, determining the amount of randomness that should be used in making decisions. However, Copycat's performance on certain problems makes it clear that more sophisticated self-watching mechanisms are needed. A salient defect of Copycat is its mindlessly loopish behavior when solving the problem "**abc ⇒ abd, xyz ⇒ ?**". As could be seen in the screen dumps on that problem, the program returns again and again to the same state of trying to take the successor of *Z* and failing. As was pointed out earlier, people too are prone to some degree of loopish behavior, but not to the extent that it occurs in Copycat, which hits the same impasse on average nine times per run on this problem. A normal human would never do this; after two or three times they would notice a pattern, and would be able to break it. But Copycat lacks mechanisms for forming, or remembering, any kind of high-level description of its behavior, or of states that it has been in before.[1] Such high-level pattern-recognition mechanisms are, in effect, analogy-making mechanisms—for example, the program would need to recognize that it was doing essentially "the same thing" each time it got stuck, even though the events leading up to the impasse might be very different each time. Thus some of the same mechanisms that Copycat uses for making analogies between letter-strings should apply to the problem of watching and responding to its own behavior. Giving Copycat such an ability would be an excellent topic for future research on this project. (Ideas about the relations among self-watching, high-level pattern recognition, and creativity are discussed in Hofstadter, 1985b.)

Another serious self-watching problem in Copycat is finding a way to have the densities of various types of codelets on the Coderack at a given time (corresponding to the various types of structures—descriptions, bonds, groups, correspondences, rules) correspond at least roughly to the kinds of structures the system current needs to build (e.g., at a given time, the program might *need* more bonds to be built in order for more progress to be made). Having too few codelets of a given type means that the program will often miss essential structures, and having too many of a given type (e.g., codelets looking for bonds) causes

---

[1] Copycat does save the exact state of the Workspace each time an impasse is hit, and the temperature remains clamped until the program decides (probabilistically) that new structures of sufficiently high quality have been built, but this is a very unsophisticated mechanism compared to the kind of high-level pattern recognition the program needs to apply to its own behavior in order to avoid being stuck in a loop.

the program to waste much of its time fruitlessly exploring again and again structures that already exist. Maintaining a proper balance in the population of codelets has emerged as an absolutely central issue in this project. This problem corresponds to a general issue that is central to high-level perception. The question is, how much time should one spend looking for new kinds of structures (and how to allocate this time among the various types of structures), and how much time should one spend concentrating on concepts that have already been identified as relevant? Again, this is the "exploration versus exploitation" trade-off. As can be seen in the screen dumps, the proper balance of bottom-up-versus-top-down pressures changes as processing proceeds: the program starts out being dominated by bottom-up forces, but as structures are built and information is gained, processing gradually shifts toward being dominated more and more by top-down pressures. The reason for this is that as more structure is built and more nodes are activated, more and more top-down scout codelets are posted, and they tend to have higher urgencies than bottom-up scout codelets, and thus gradually come to dominate on the Coderack in terms of number and urgency.

There are two balancing problems here: the balance between bottom-up and top-down forces, and the balance among codelets looking for the various types of structures. The method just described for achieving a good bottom-up-versus-top-down balance in Copycat works fairly well, and emerges naturally from other mechanisms in the system. However, it proved more difficult to develop ways of maintaining a reasonable balance among codelets looking for various types of structures (e.g., at a given time, should the program spend more time looking for groups than for bonds?). In order to achieve such a balance, the system requires self-watching mechanisms to determine what types of codelets it currently needs. Temperature is such a mechanism, but in the current version of Copycat, temperature was not enough to solve this problem of codelet balance; more detailed self-watching mechanisms seem to be needed. In the current program, I have added a somewhat imperfect mechanism to help achieve a reasonable balance: when an attempt is made to post codelets (bottom-up or top-down) corresponding to a particular type of structure (e.g., bonds), the program first makes a rough assessment of the current need for that type of structure in the problem by looking more specifically (i.e., more specifically than is done when calculating temperature) at the causes of the unhappiness of objects in the problem (e.g., do many objects lack bonds to their neighbors?). The program then decides probabilistically, based on this assessment, whether or not such codelets should be allowed to be posted (e.g., if many objects lack bonds,

then it is likely that bond scouts will be allowed to be posted). This filtering mechanism works fairly well, but it is unsatisfactory in that it is too global and centralized, and thus goes against the philosophy of local and distributed processing underlying Copycat. Self-watching is essential, but it should be done in a less centralized way than in the current version of the program.

The weaknesses discussed in this chapter are by no means the only problems with the program; many more exist at various levels of detail. But the problems of top-down forces, focus of attention, and self-watching are currently the most salient and interesting problems, and are, I think, the issues most relevant to modeling high-level perception in general. They are the problems that should probably have highest priority in future work on this project.

CHAPTER VII

RESULTS OF SELECTED "LESIONS" OF COPYCAT

In this chapter I give the results of six experiments designed to elucidate the roles played by various aspects of the program's architecture. The purpose of doing these experiments was to further illustrate how the program works and to demonstrate the reason for the presence of certain architectural features by showing what happens when they are "lesioned" (i.e., removed or altered). In each experiment, the program was altered in some way, and was then run 200 times on one or more of the five target problems.

7.1 Experiment 1: Suppression of Terraced Scanning

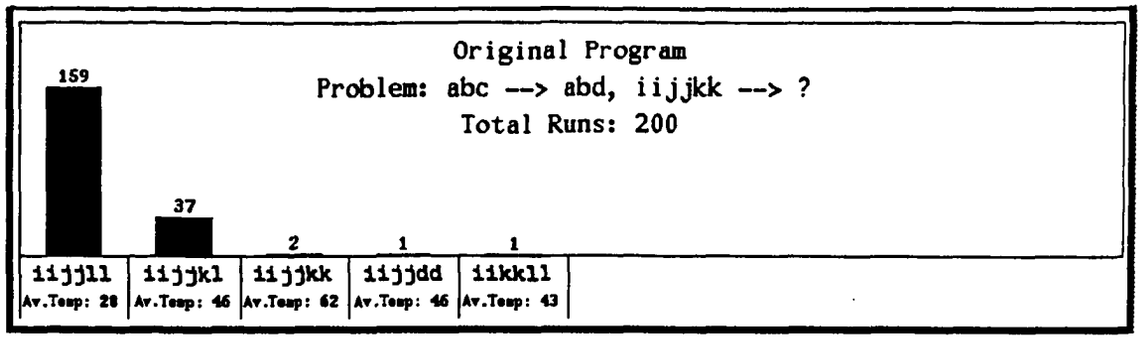
Recall that in Copycat, a structure is built by a chain of codelets,

scout ⇒ strength-tester ⇒ builder

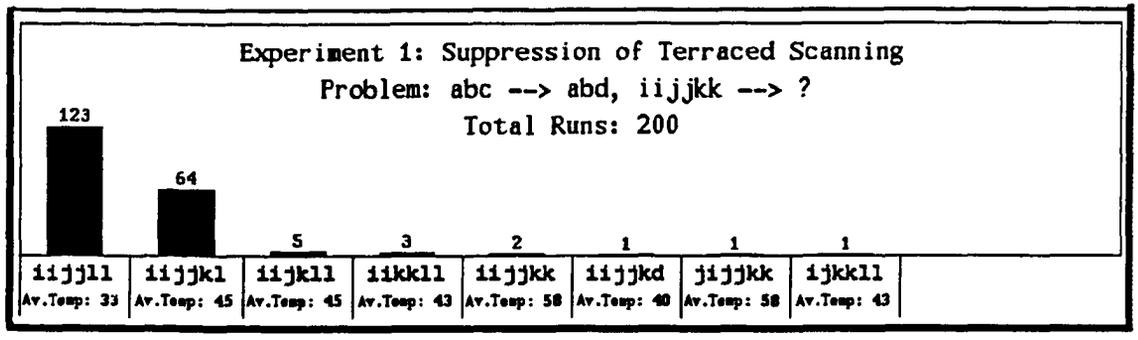
rather than by a single codelet. The purpose of this experiment was to examine the role played by this breaking-up of the process of structure-building. For this experiment, the usual chain was compressed into a single codelet: the program was modified so that a single codelet carried out all three tasks (scouting out a possible structure, testing its strength, and if the structure was found to be strong enough, building it). The same types of scout codelets as in the original program were present here; the difference was that rather than posting follow-up codelets, each scout carried out all three tasks.

I ran this experiment on two problems: "**abc ⇒ abd, iijjkk ⇒ ?**" and "**abc ⇒ abd, mrrjjj ⇒ ?**".

The following bar graph gives the results for 200 runs of the *original* (unmodified) program on the first problem:



The following bar graph gives the results for 200 runs of the *modified* program on the first problem:



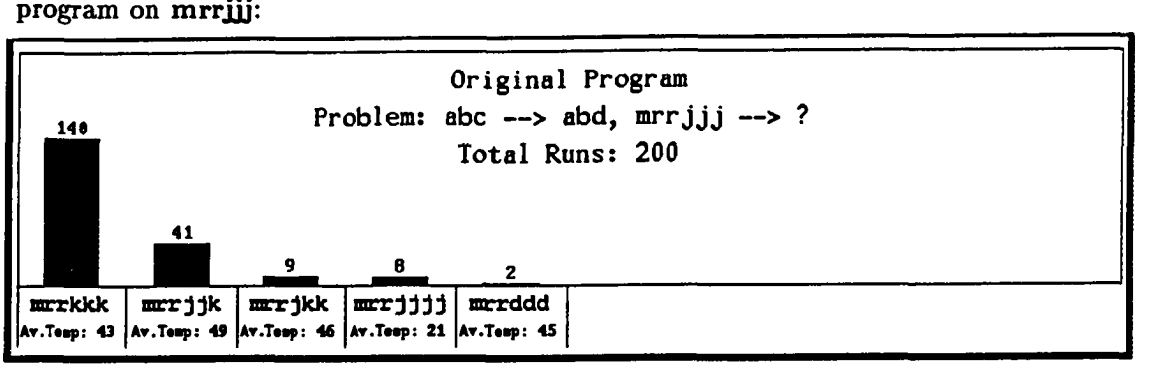
As can be seen, the modified program produces a larger number of badly justified fringe answers than the original program. This is because structures are built much more quickly in the modified version: a structure is built in one monolithic step rather than having to wait after each separate step for the next codelet in the chain to be chosen to run. Exploration of structures becomes all-or-nothing: if a structure is explored at all, it is fully evaluated all at once, as opposed to what happens in the original program, in which the further exploration of promising structures is given high urgency and tends to proceed quickly, while the further exploration of weak structures is given low urgency and tends to proceed slowly. Thus in the modified program, the parallel terraced scan of possibilities loses some of its parallel and terraced nature.[1]

In the modified version, most weak structures still fail to pass the strength test and are not built, but some, whose exploration would ordinarily be crowded out by other, higher-

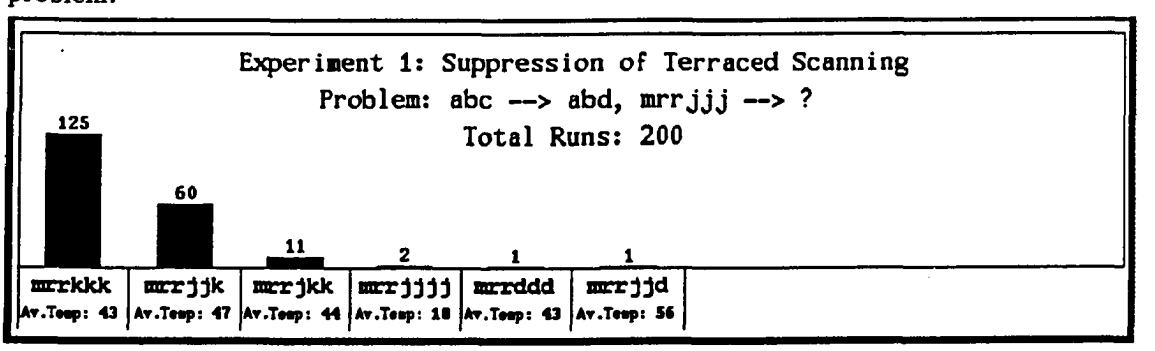
[1] However, it is not lost entirely: even though *individual* structures are no longer considered and built in a parallel terraced manner, the program still carries out a parallel terraced scan of coherent *collections* of structures. Once certain structures are built (e.g., a new successor bond or a new correspondence), the resulting changes in the state of the Slipnet and the Workspace lead to top-down codelets and new structure-strength values that increase the likelihood and speed of exploring compatible and supporting structures.

urgency explorations, are built nonetheless, and they then affect the building of subsequent structures. The effects on the program's behavior are statistical, and can be seen in the bar graph above. The modified program produces about three times as many fringe answers as the original, and almost twice as many instances of **iijjkl**, showing that statistically speaking, strong structures are not being built as often, and weak structures (such as those leading to the fringe answers) are being built and are surviving more often than in the original program.

The following bar graph gives the results for 200 runs of the *original* (unmodified) program on **mrrjjj**:



The following bar graph gives the results for 200 runs of the *modified* program on this problem:



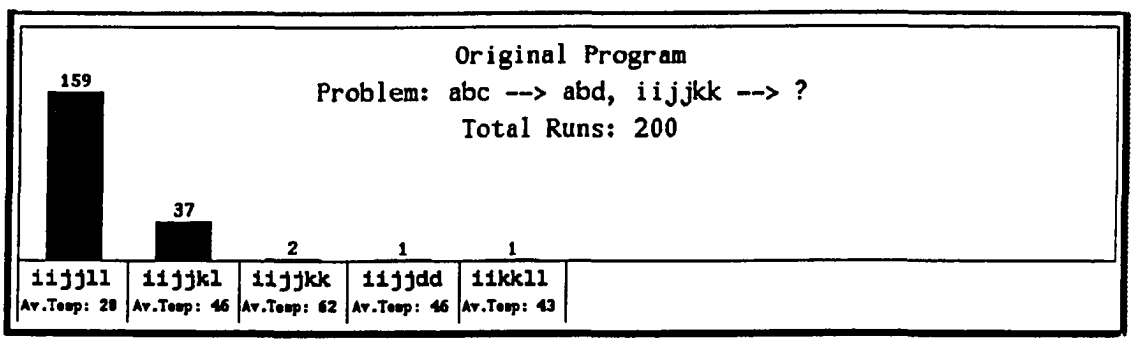
The results here are similar to those for **iijjkk**. There are about one-and-a-half times as many instances of **mrrjjk** as in the original, showing that strong structures (such as the **kkk** group) were not built as often. Perhaps most significantly, there are only one-fourth as many instances of **mrrjjjj** as in the original. As could be seen in the screen dumps in Chapter 4, a careful, terraced exploration of possible structures is important for arriving at this answer, and the statistics here back this up.

7.2 Experiment 2: Suppression of Breaker Codelets

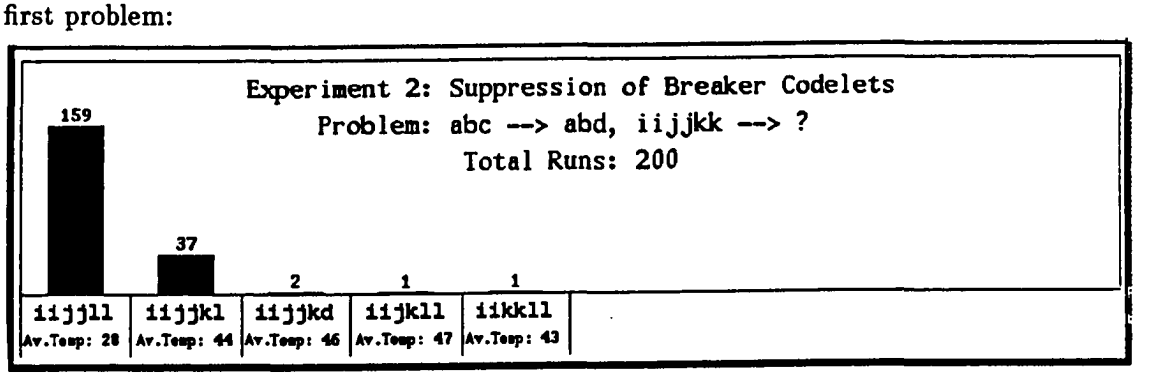
For this experiment, breaker codelets were taken out of the program; everything else remained the same.

I ran this experiment on two problems: "**abc ⇒ abd, iijjkk ⇒ ?**" and "**abc ⇒ abd, xyz ⇒ ?**".

The following bar graph gives the results for 200 runs of the *original* (unmodified) program on the first problem:

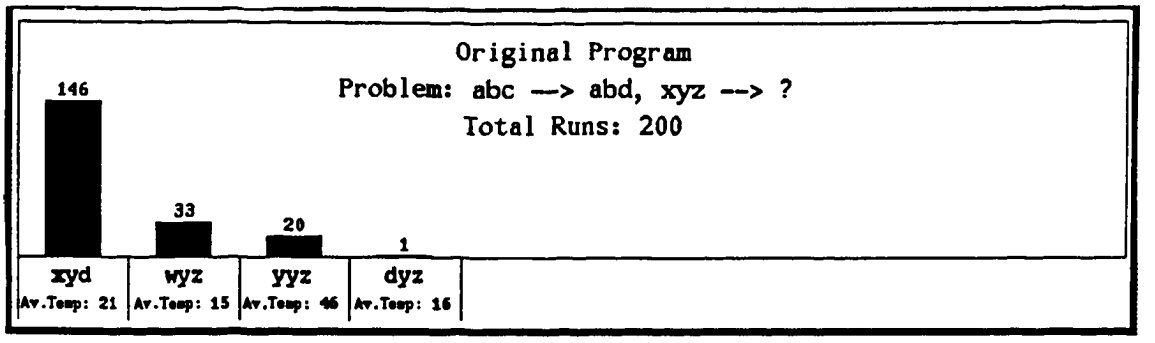


The following bar graph gives the results for 200 runs of the *modified* program on the first problem:

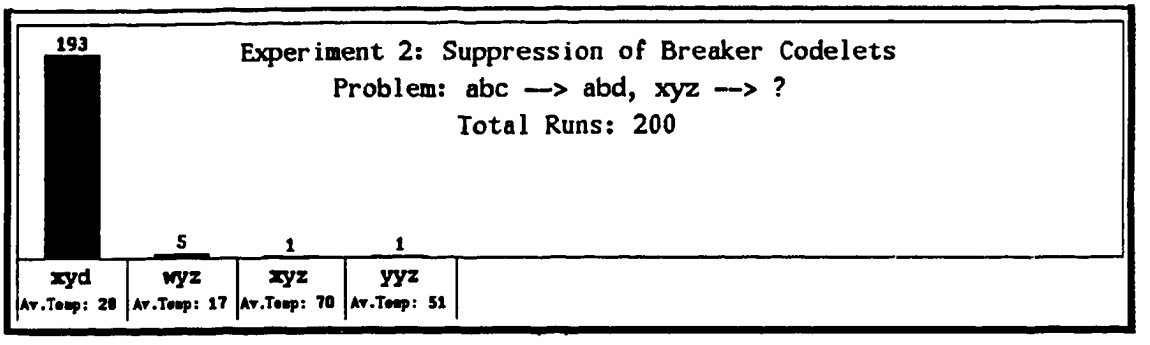


As can be seen by comparing the bar graphs, the absence of breaker codelets had virtually no effect on the program's performance here. The frequencies of the main two answers are the same (the exact equality is a coincidence) and the number of fringe answers is the same (though the set of fringe answers is slightly different in each case). This is not surprising, since breaker codelets, which tend to run only at high temperatures, do not play much of a role in a problem like this, in which the temperature falls fairly quickly. The average time to produce an answer was roughly the same in the two cases (572 codelet steps in the modified version versus 589 in the original).

The following bar graph gives the results for 200 runs of the *original* (unmodified) program on **xyz**:



The following bar graph gives the results for 200 runs of the *modified* program on this problem:



Here there is a significant difference in performance, illustrating the role of breaker codelets in this problem. There are almost 7 times as many instances of **wyz** in the original as in the modified version. Without codelets to break structures at high temperature, it is extremely difficult to escape from the impasse of trying to take the successor of *Z*, since the **c–z** correspondence is very strong and is supported by other strong structures. This is the case even though decisions are more random at high temperatures; thus structure-breaking codelets are a very important mechanism for escaping from impasses. Without them, the program's only escape from the snag is, in most cases, to restructure the rule from "Replace rightmost letter by successor" to "Replace rightmost letter by *D*", and to answer **xyd**.

Interestingly, the program without breaker codelets tends to arrive at an answer much more quickly than the original program (1644 codelets on average versus 3218 in the original). This is because when breaker codelets are suppressed, the program does not have to spend time building new structures to replace structures that have been broken.

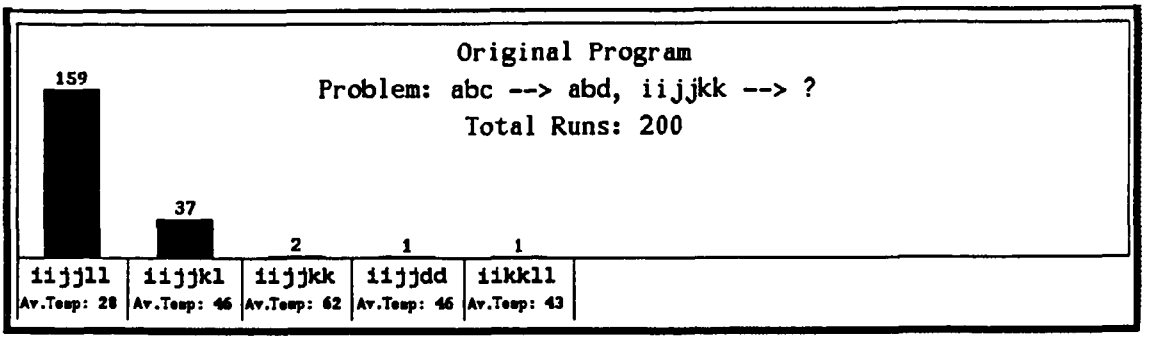
7.3 Experiment 3: Suppression of Different Conceptual-Depth Values

As was described earlier, the conceptual-depth values in the Slipnet play a number of roles. The conceptual-depth value of a given node affects the node's rate of activation decay, the

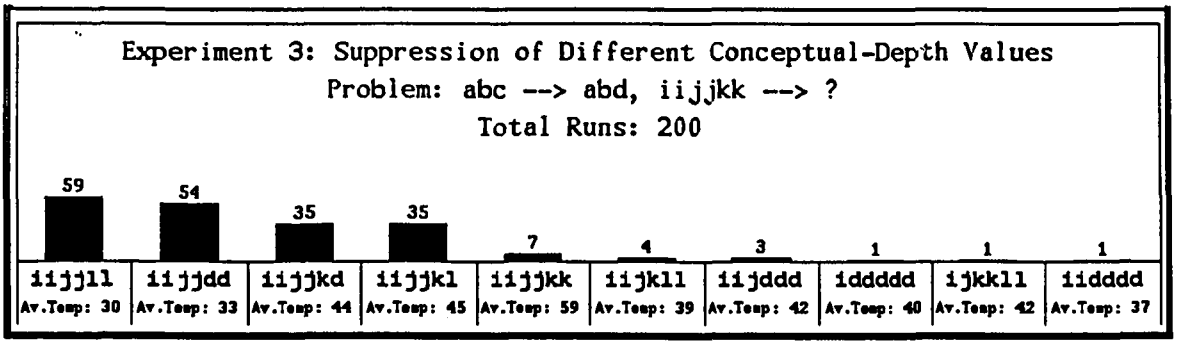
urgencies of top-down codelets posted by that node, the strength of descriptions involving that node, the probability of making a slippage involving that node, the strength of concept-mappings involving that node, and the probability of a rule-building codelet choosing that node as part of a rule. For this experiment, all nodes in the Slipnet were given equal conceptual-depth values (each was given a value of 50—see Appendix B for the original values). Everything else remained the same.

I ran this experiment on "**abc ⇒ abd, iijjkk ⇒ ?**" and "**abc ⇒ abd, mrrjjj ⇒ ?**".

The following bar graph gives the results for 200 runs of the *original* (unmodified) program on the first problem:



The following bar graph gives the results for 200 runs of the *modified* program on the first problem:



As can be seen from the bar graphs, making the conceptual-depth values all equal had a dramatic effect on the program's performance. The most striking difference here is the increase in answers derived from the rule "Replace rightmost letter [or group] by *D*". This is to be expected, since the "*D*" rules are now just as strong as the "*rightmost letter* [or *group*]" rules. There are also more instances of **iijjkk**, based on "Replace *C* by *D*", though the total number is still small. This rule is now just as strong as the other two rules, but the fact that the **c** is usually seen as corresponding to the rightmost letter or group in the target string prevents "Replace *C* by *D*" from being built very often. A *rightmost* ⇒ *rightmost* correspondence is asserting, in effect, that the **c** should be viewed as "the rightmost letter",

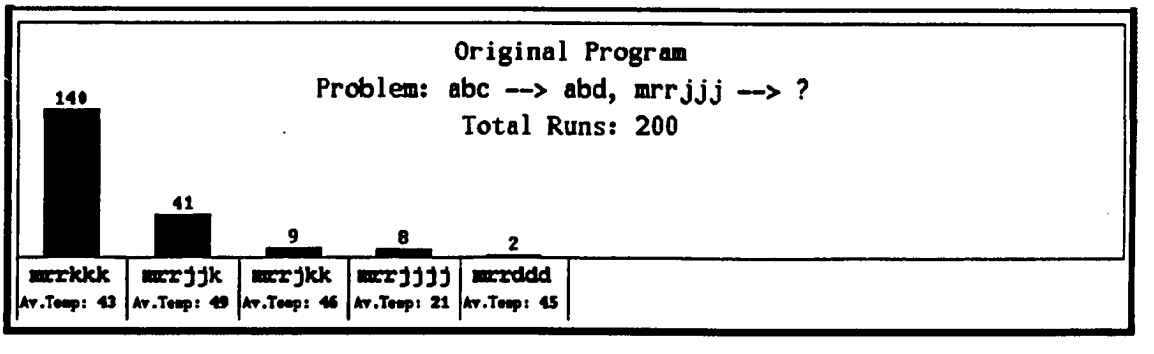
whereas the "Replace *C* by *D*" rule is asserting that the c should be viewed as "a *C*". These views are incompatible, so in order to be built, this rule would have to fight with and defeat the *rightmost* ⇒ *rightmost* correspondence. This puts it at a disadvantage with respect to the other possible rules.

Another difference is that here there are almost twice as many instances of answers for which the rightmost *letter*, rather than the rightmost *group*, is replaced: this is because the urgencies of most top-down codelets are not as high as in the original, so mutually-supporting sameness groups are not explored or built as often or as quickly.

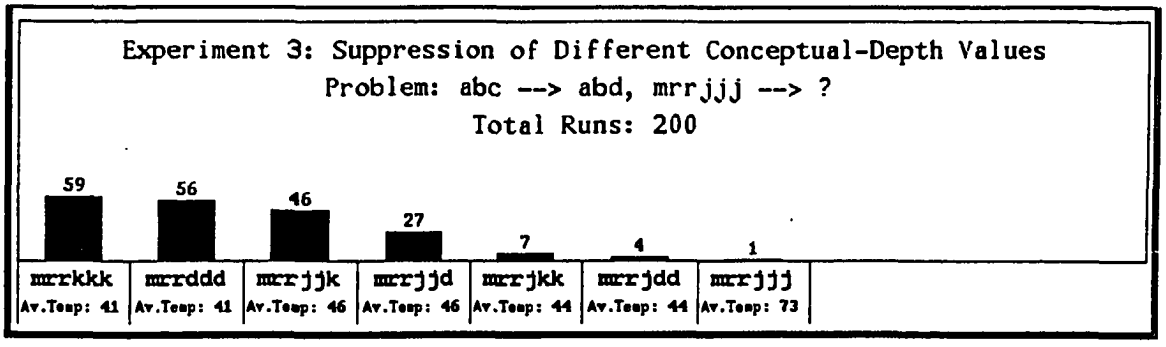
In general, there is less pressure from top-down codelets not only because their urgency is lower, but also because the activation in nodes (such as *sameness* and *sameness-group*) that originally had greater conceptual depth now tends to decay much more quickly (and conversely, nodes that were originally of lesser conceptual depth now stay active longer) than in the original program, so not as many top-down codelets are posted. Thus good structures do not get built as fast, and the temperature stays higher longer. This helps to increase the number of bad-grouping answers (many of which now involve replacing the letters in the bad group by d's rather than by their successors, as in the frighteningly blockheaded answer **iddddd**).

The reduced force (in terms of both urgency and number) of top-down codelets, along with the fact that the temperature stays higher longer, means that on average it takes the program longer to get to an answer. The average number of codelets run in the modified program is 743, versus 589 in the original.

The following bar graph gives the results for 200 runs of the *original* (unmodified) program on **mrrjjj**:



The following bar graph gives the results for 200 runs of the *modified* program on this problem:



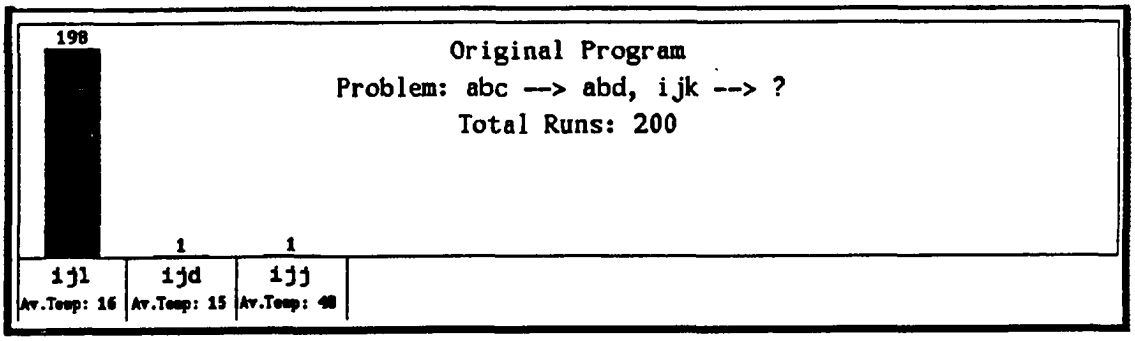
The effects here are similar to those for **iijjkk**. Notice that the modified program never once produced **mrrjjij** during the 200 runs, whereas it was produced 8 times in the original program's 200 runs. This shows the necessity of strong top-down forces for arriving at this answer (strong top-down forces are needed to create a single-letter-group and to notice and build bonds among group-lengths). Such top-down forces are significantly reduced in the modified program.

7.4 Experiment 4: Suppression of Dynamic Link-Lengths

Recall that in Copycat, links in the Slipnet shrink in length when the node labeling them is active. For example, when *opposite* is active, all *opposite* links (e.g., the link between *leftmost* and *rightmost*) shrink. For this experiment, the program was modified so that link-lengths were no longer dynamic: links always remained at their original length.

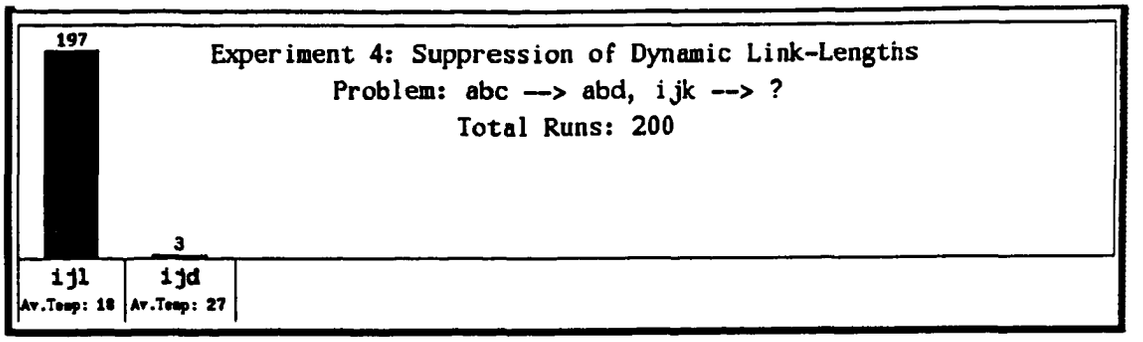
I ran this experiment on three problems: "**abc ⇒ abd, ijk ⇒ ?**", "**abc ⇒ abd, kji ⇒ ?**", and "**abc ⇒ abd, xyz ⇒ ?**".

The following bar graph gives the results for 200 runs of the *original* (unmodified) program on the first problem:



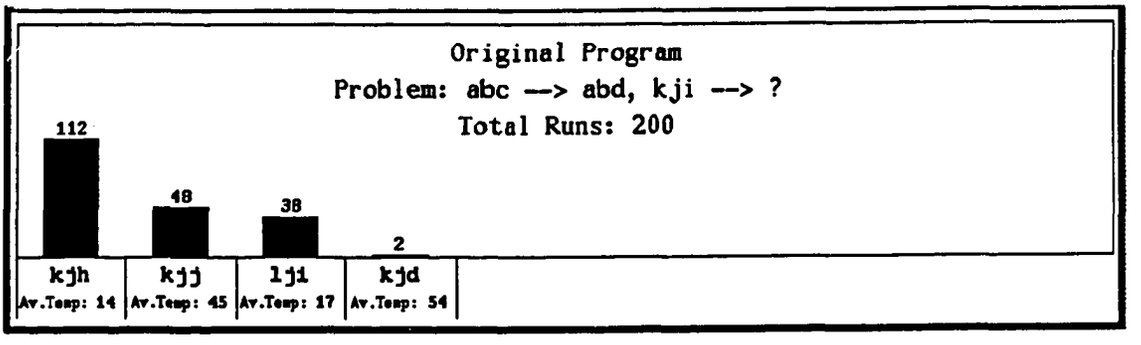
The following bar graph gives the results for 200 runs of the *modified* program on the

first problem:

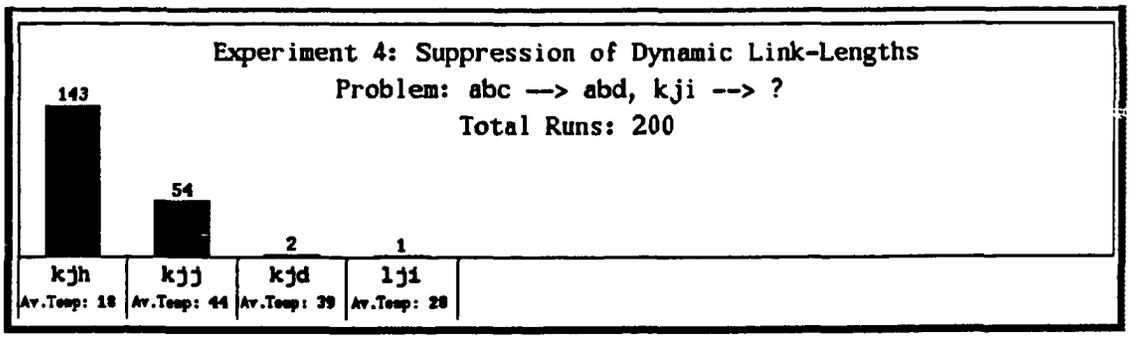


As can be seen, the modification had basically no effect on the relative frequencies of answers to this problem. There is a difference here, however: the modified program took slightly longer on average to arrive at an answer (329 versus 289 codelets run on average). It is slower because dynamic link-lengths can act as a top-down force: when the concept *successor*, say, becomes active, this causes successor links (e.g., between A and B or I and J) to shrink, and these relationships to thus be seen as closer. This speeds up the building of successor bonds since the bonds are judged to be stronger.

The following bar graph gives the results for 200 runs of the *original* (unmodified) program on kji:



The following bar graph gives the results for 200 runs of the *modified* program on this problem:



Here there is a visible difference in the two bar graphs: the number of kjh instances

goes way up and the number of **lji** instances plummets dramatically in the modified version. The reason for this is as follows. The program answers **kjh** when it has made *vertical* correspondences (*leftmost* ⇒ *leftmost* and *rightmost* ⇒ *rightmost*) between **abc** and **kji**. As was shown in the screen dumps in Chapter 4, these correspondences force a view in which one string is seen as a successor group and the other as a predecessor group. In this case, when a whole-string mapping is made between **abc** and **kji**, the slippage *successor* ⇒ *predecessor* is made automatically (without requiring that the link between *successor* and *predecessor* already be shrunk); the slippage is forced by the whole-string mapping, and *opposite* is activated only after the slippage is made. On the other hand, the answer **lji** is harder for the program (even the original version) to get. As was seen in Chapter 4, it is produced when the program views **abc** and **kji** as moving in the same alphabetic direction but in different spatial directions. This view produces a whole-string mapping with the slippage *right* ⇒ *left*, which activates *opposite*. Only then, with links between *opposite* nodes being shrunk, is it likely that *diagonal* correspondences (*leftmost* ⇒ *rightmost* and *rightmost* ⇒ *leftmost*) will be built. (These two concept-mappings, although closely related to *left* ⇒ *right* and *right* ⇒ *left*, do not come about automatically when the latter two have been made. They must be made independently, although their construction is strongly facilitated by the latter two concept-mappings.)

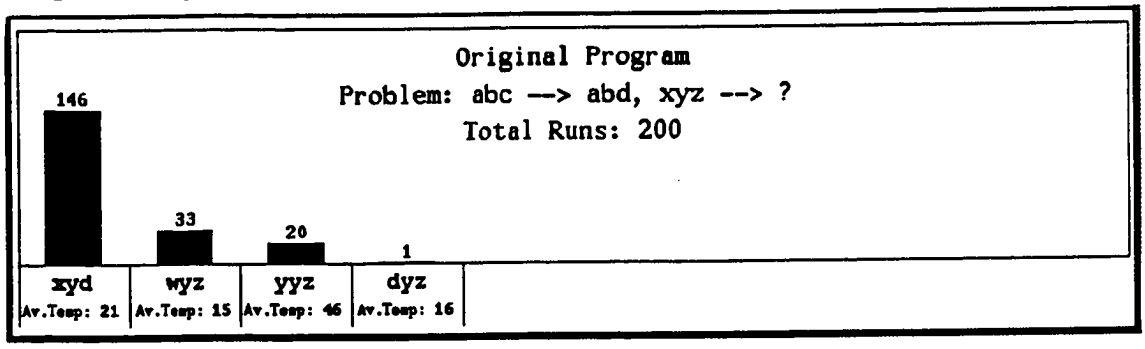
Thus the answer **lji** relies on dynamic link-lengths, whereas **kjh** does not. The difference is that the slippage needed for the latter (*successor* ⇒ *predecessor*) is made automatically when **abc** and **kji** (viewed in opposite alphabetic directions) are mapped as wholes, whereas the slippage needed for the former (*rightmost* ⇒ *leftmost*) can be easily made only after a succession of events has taken place: the whole-string slippage (*right* ⇒ *left*) is made, *opposite* is activated, and links between opposite nodes are shrunk.

It is possible that this asymmetry in the routes to the two answers is not psychologically realistic, even though people tend to answer **kjh** more often than **lji**. It seems plausible that once the slippage *right* ⇒ *left* is made, the closely related slippage *rightmost* ⇒ *leftmost* should come immediately on its "coattails", not merely as a result of the activation of *opposite*. However, the current version of Copycat has no mechanism implementing such a "coattails" effect (see Hofstadter, Mitchell, & French, 1987, for a more detailed discussion of how this effect might work in Copycat).

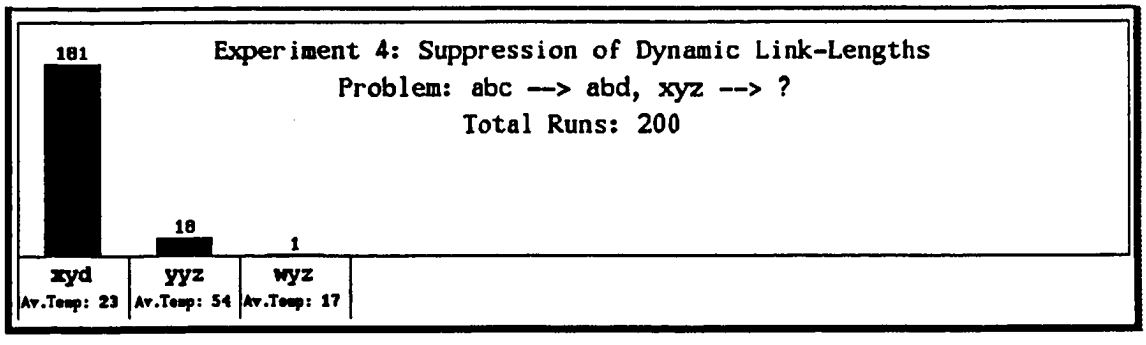
For this problem, the average time taken to arrive at an answer was not very different in the modified version and the original version: 375 codelets run on average in the modified

version versus 387 in the original. The reason for this was that even though the modified version is intrinsically slower (as was seen on **ijk** above), it takes longer for the program to come up with the answer **lji** than **kjh**, so in the modified version the intrinsic slowness was balanced by the reduction in instances of answer **lji**.

The following bar graph gives the results for 200 runs of the *original* (unmodified) program on **xyz**.



The following bar graph gives the results for 200 runs of the *modified* program on this problem:



Here the number of instances of **wyz** goes way down, and the number of instances of **xyd** goes up. The answer **wyz** is very hard to make without dynamic link-lengths. In the original program, once the **a–z** correspondence—with concept-mappings *first* ⇒ *last* and *leftmost* ⇒ *rightmost*—is built, *opposite* becomes active, making all *opposite* links shorter, and hence making it more likely for the **c–x** correspondence to be built and for **abc** and **xyz** to be seen as going in opposite spatial and alphabetic directions. Without dynamic link-lengths, it is much harder for all these mutually supporting structures to be built, which means that the unreinforced **a–z** correspondence is so weak that it tends to be broken quickly. This is why the answer **xyd** is overwhelmingly prevalent in the modified program. The answer **yyz** comes from building only the **a–z** correspondence (with slippage *leftmost* ⇒ *rightmost*), without being able to make a whole-string mapping with a *successor* ⇒ *predecessor* slippage. As was the case for **ijk**, the modified program takes longer to come up with an answer: on

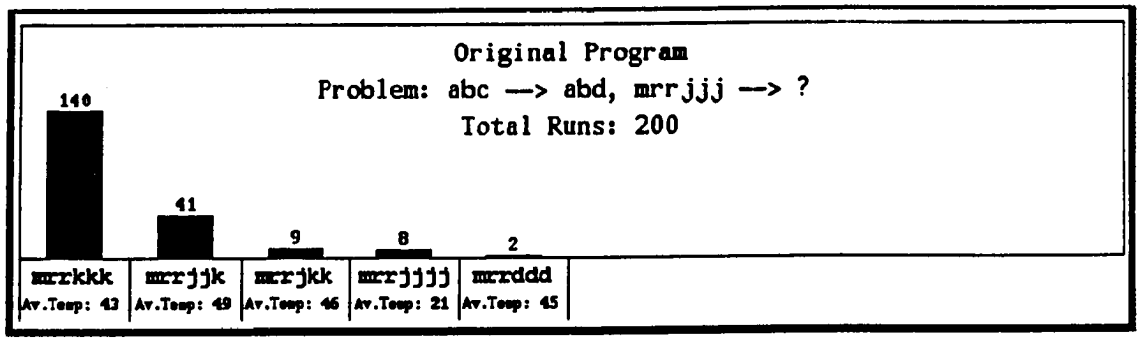
average 4468 codelets ran here versus 3322 in the original.

7.5 Experiment 5: Clamping Temperature at 100

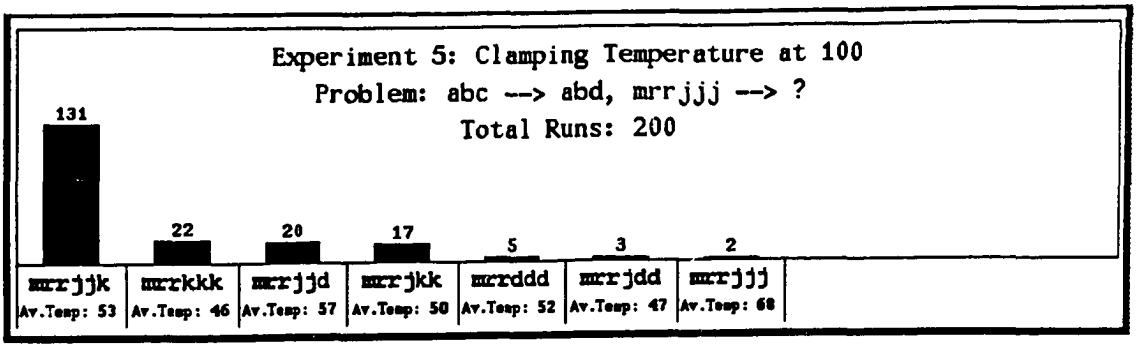
The point of this experiment was to see the effect of a persisting high temperature on the program's performance. In this experiment, the amount of randomness to use in making probabilistic decisions was fixed at its 100-degree value (recall that at a temperature of 100, decisions are made with a high degree of randomness, though they are still not *uniformly* random).

One problem with carrying out such an experiment is, if the temperature is always 100, rule-translator codelets will essentially never decide that a sufficient amount of good structure has been constructed in order to translate the rule and allow an answer to be built. To take care of this problem, a separate value for temperature was maintained, calcualted as in the original program as a function of the happinesses of the objects in the Workspace. This "real" temperature was visible only to rule-translator codelets. For all other purposes, the temperature was clamped at 100.

The following bar graph gives the results for 200 runs of the *original* (unmodified) program on "**abc ⇒ abd, mrrjjj ⇒ ?**".



The following bar graph gives the results for 200 runs of the *modified* program on "**abc ⇒ abd, mrrjjj ⇒ ?**". (The average final temperature displayed here corresponds to the "real" temperature values that were visible only to rule-translator codelets).



As can be seen, the modified program's performance is vastly different from that of the original program. As was discussed in Chapter 3 and illustrated in Chapter 4, temperature affects almost every aspect of the program, and it can be seen from the bar graph above that a persisting high temperature tends to prevent a coherent set of structures from being built. The answer **mrrjjk** dominates here, and the more structured **mrrkkk** is much less likely to be given. Answers that were on the fringes for the original program (e.g., **mrrjjd** and **mrrjjj**) are much more likely to be given here. Even if the program stumbles onto a good pathway, the high amount of randomness here makes it impossible for the program's resources to shift to exploring that pathway. The answer **mrrjjjj** was never given during the 200 runs; since the temperature stays high, the necessary top-down forces never get the chance they need to construct the subtler structures required for this answer.

It can be seen that the average final temperatures for the modified program (corresponding to the "real" temperature values, as described above) are all higher than the corresponding temperatures for the original program, reflecting the fact that on average, not as much strong structure was constructed here.

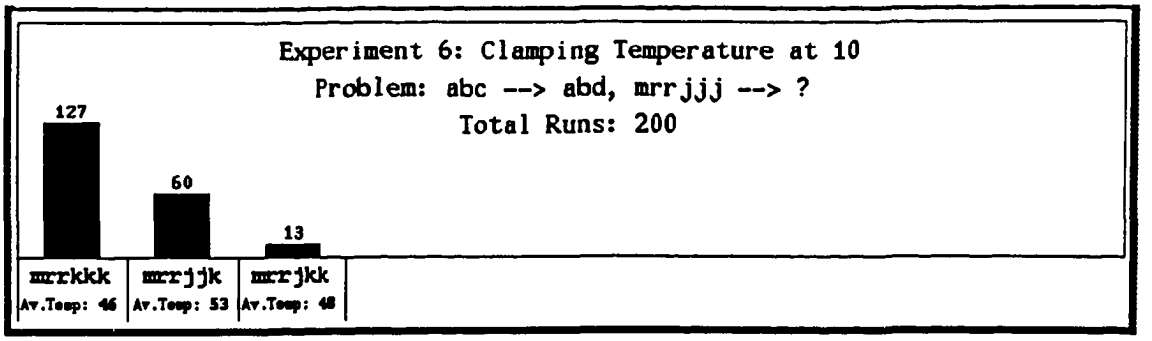
Since the persisting high temperature makes it hard for a coherent set of structures to be built, the modified program is much slower at coming up with answers than the original: on average 1130 codelets ran versus 846 in the original.

7.6 Experiment 6: Clamping Temperature at 10

Here the temperature was clamped at a very *low* value (10) for all purposes except deciding when to translate the rule (as in Experiment 5, temperature was calculated as usual for use by rule-translator codelets, and the average final values of these real temperatures are displayed in the bar graph).

The following bar graph gives the results for 200 runs of the *modified* program on

"**abc ⇒ abd, mrrjjj ⇒ ?**".



In Experiment 5, decisions were made too randomly. Here the opposite effect takes place: the low temperature means that decisions are made very deterministically (e.g., at any time, the highest-urgency codelet is almost certain to be chosen next, the strongest structure in a competition is almost certain to win, etc.), even when very little structure has been built. Again, this has striking effects on the program's performance. Unlike in the previous experiment, here the answer **mrrkkk** dominates, but even so, there are still more instances of **mrrjjk** here than in the original. The most striking difference is the lack of fringe answers here (though there are several instances of **mrrjkk** which, as usual, results from the program's occasional grouping problems). Since the modified program is now quite deterministic, weaker rules (e.g, "Replace rightmost letter by *D*" or "Replace *C* by *D*") never prevail. This modification makes the program quite conservative, so it doesn't produce as many farfetched weak answers (such as **mrrddd**), but it also never (in 200 runs) came up with **mrrjjjj**, which requires the exploration of some riskier routes. The high degree of determinism means that what appears to be the best possibility gets almost all of the program's attention at any given time, so less-obvious structures, such as single-letter groups, length descriptions, and bonds between group-lengths, are much less likely to be considered in any depth. The fact that, at any given time, the program tends to focus almost *all* of its resources on what it sees as the most promising avenue turns out to be a waste of time, since the program tends to explore the same strong structures again and again. This is why there are more instances of **mrrjjk** than in the original program: the program might, for example, spend too much time exploring again and again the possibility of building the very strong **abc** whole-string group even after it has already been built, and never get around to building the **jjj** group before it decides to produce an answer.

Unlike in Experiment 5, the modified program here is much faster than the original at reaching an answer (468 codelets on average versus 846 in the original). One reason is

similar to that in Experiment 1: the high degree of determinism results in a quite serial form of exploration, in which seemingly good structures are explored and built very quickly, while seemingly weaker structures are hardly explored at all, even at very early stages. Thus the program doesn't spend time exploring many possibilities, and can come to an answer much more quickly. But again, the trade-off is that certain possibilities (such as building single-letter groups or noticing group-lengths) are in effect completely excluded from the start, and the program is thus liable to miss interesting but not immediately obvious ways of interpreting the situations (as it did here). This shows the necessity for a balance between exploitation and exploration that was discussed earlier; in Experiment 5, the program erred on the exploration side, and here it errs on the exploitation side. A belief underlying this model (and supported by the solution to the two-armed bandit problem discussed earlier) is that not only is a balance needed, but there must be a smooth and gradual transition from a more random and parallel *exploration* mode in early stages to a more deterministic and serial *exploitation* mode in later stages when the system has more information upon which to base decisions.

7.7 Summary

The experiments described in this chapter have further illustrated the roles played by certain architectural features of Copycat: the role of breaking up structure-building into chains of codelets (i.e., the *terraced* scan), the role of structure-breaking codelets, the role of conceptual-depth values in the Slipnet, the role of dynamic link-lengths in the Slipnet, and the role of temperature. There are many more such experiments that could be done (e.g., removing all bottom-up or all top-down codelet types, running the program with no spreading activation in the Slipnet, limiting the Coderack to various different sizes, and so on), and in general it would be very interesting to systematically vary the parameters and formulas in the system and to observe the effects on Copycat's behavior. This experiments described in this chapter represent a first step in this longer-term process of exploring the effects of such variations on the model.

CHAPTER VIII

COMPARISONS WITH RELATED WORK

In this chapter I compare the Copycat project with other research on computer models of analogy-making and with some other artificial-intelligence architectures that are related to Copycat's architecture. I also discuss Copycat's place in the spectrum of computer models of intelligence, which ranges from high-level symbolic models to low-level subsymbolic models.

8.1 Comparisons With Other Research on Analogy-Making

A fair amount of research has been done in artificial intelligence and cognitive science on constructing computer models of analogy-making, almost all of it concentrating on the use of analogical reasoning in problem-solving. Most of these models concentrate on how a mapping is made from a source problem whose solution is known to a target problem whose solution is desired, with some kind of representation of the various objects, descriptions, and relations in the source and target problems given to the program ahead of time. Very few computer models focus (as Copycat does) on how the *construction* of representations for the source and target situations interacts with the mapping process, and how new, previously unincluded concepts can be brought in and can come to be seen as relevant in response to pressures that emerge as processing proceeds. In short, very few computer models of analogy-making are models of high-level perception, concepts, and conceptual slippage in the way Copycat is.

In this section, rather than giving a complete survey of computer models of analogy-making, I will discuss (and compare with Copycat) in detail three different projects, chosen for their prominence in artificial intelligence and for their relevance with respect to the Copycat project. This leaves out a discussion of many other models of analogy-making less

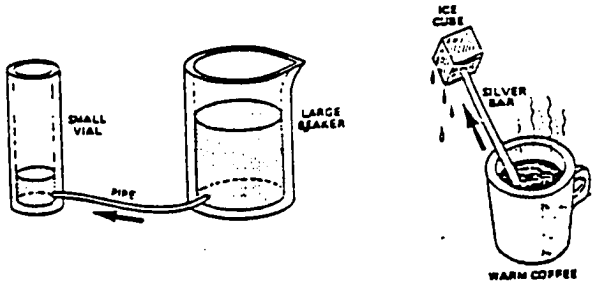


Figure 8.1: Water-flow and heat-flow situations (from Falkenhainer, Forbus, & Gentner, 1989).

related to Copycat; a good number of these are described by Hall (1989) and by Kedar-Cabelli (1988a).

8.1.1 Gentner et al.

Dedre Gentner's research is perhaps the best-known work in cognitive science on analogy. She has formulated a theory of analogical mapping, called the "structure-mapping" theory (Gentner, 1983), and she and her colleagues have constructed a computer model of this theory: the Structure-Mapping Engine, or SME (Falkenhainer, Forbus, & Gentner, 1989). The structure-mapping theory describes how mapping is carried out from a source situation to a (sometimes less familiar) target situation. The theory gives two principles for analogical mapping: 1) *relations* between objects rather than *attributes* of objects are mapped; and 2) relations that are part of a coherent interconnected system are preferentially mapped over relatively isolated relations (the "systematicity" principle). Gentner's definition of analogy in effect presupposes these mapping principles. According to her, there is a continuum of kinds of comparison: an "analogy" is a comparison in which only systematic relations are mapped, whereas a comparison in which both attributes and relations are mapped is a "literal similarity", not an analogy. I do not make such a sharp distinction, as can be seen from the spectrum of examples of analogy-making given in Chapter 1.

One of Gentner's examples of an analogy is illustrated in Figure 8.1 (from Falkenhainer, Forbus, & Gentner., 1989). The idea "heat flow is like water flow" is illustrated by mapping a situation in which water flows from a beaker to a vial through a pipe onto a situation in which heat flows from coffee in a cup to an ice cube through a metal bar.

The predicate-logic representations given for these two situations are displayed in Fig-

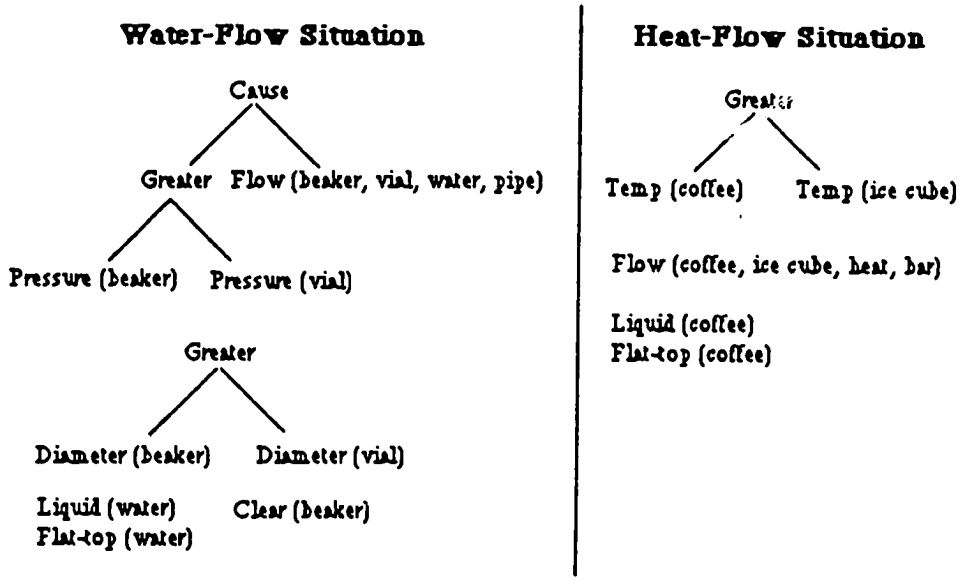


Figure 8.2: The predicate-logic representations for the water-flow and heat-flow situations (from Falkenhainer, Forbus, & Gentner, 1989).

ure 8.2. The idea is that the causal relation tree on the left (representing the fact that greater pressure in the beaker causes water to flow from the beaker to the vial through the pipe) is a systematic structure and should thus be mapped to the heat-flow situation, whereas the other facts ("the diameter of the beaker is greater than the diameter of the vial", "water is a liquid", "water has a flat top", etc.) are irrelevant and should be ignored. Ideally, mappings should be made between *pressure* and *temperature*, *coffee* and *beaker*, *vial* and *ice cube*, *water* and *heat*, *pipe* and *bar*, and more obviously, *flow* and *flow*. Once these mappings are made, a conjecture about the cause of heat flow in the situation on the right can be made by analogy with the causal structure in the situation on the left. Gentner claims that if people recognize that this causal structure is the deepest and most interconnected system for this analogy, then they will favor it for mapping.

Gentner gives the following (possibly conflicting) criteria for judging the quality of an analogy: 1) *clarity*—a measure of how clear it is which things map onto which other things; 2) *richness*—a measure of how many things in the source are mapped to the target; 3) *abstractness*—a measure of how abstract the things mapped are, where the degree of "abstractness" of an attribute or relation is its "order": attributes (e.g., "flat-top" in the exam-

ple above) are of the lowest order, relations whose arguments are objects or attributes (e.g., "flow") are of higher order, and relations whose arguments are relations (e.g., "cause") are of even higher order; and 4) *systematicity*—the degree to which the things mapped belong to a mutually constraining conceptual system.

The computer model of this theory (SME) takes a predicate-logic representation of two situations (such as the representation given in Figure 8.2), makes a mapping between objects, attributes, and relations in the two situations, and then makes inferences from this mapping (such as "the greater temperature of the coffee causes heat to flow from the coffee to the ice cube"). The only knowledge the program has of the two situations is their syntactic structures (e.g., the tree structures given for the water-flow and heat-flow situations given above); it has no knowledge of any kind of semantic similarity between various descriptions and relations in the two situations. All processing is based on syntactic structural features of the two given representations.

SME first uses a set of "match rules" (provided to the program ahead of time) to make all "plausible" pairings between objects (e.g., *water* and *heat*) and between relations (e.g., *flow* in the case of water and *flow* in the case of heat). Examples of such rules are: "If two relations have the same name, then pair them"; "If two objects play the same role in two already paired relations (i.e., are arguments in the same position), then pair them"; "Pair any two functional predicates" (e.g., *pressure* and *temperature*). It then gives a score to each of these pairings, based on factors such as: Do the two things paired have the same name? What kind of things are they (objects, relations, functional predicates, etc.)? Are they part of systematic structures? The kinds of pairings allowed and the scores given to them depend on the set of match rules given to the program; different sets can be supplied.

Once all plausible pairings have been made, the program makes all possible sets of consistent combinations of these pairings, making each set (or "global match") as large as possible. "Consistency" here means that each element can match only one other element, and a pair (e.g., *pressure* and *temperature*) is allowed to be in the global match only if all the arguments of each element are also paired up in the global match. Consistency ensures clarity of the analogy, and the fact that the sets are maximal shows a preference for richness. After all possible global matches have been formed, each is given a score based on the individual pairings it is made up of, the inferences it suggests, and its degree of systematicity. Gentner and her colleagues have compared the relative scores assigned by the program with the scores people give to the various analogies (Skorstad, Falkenhainer,

& Gentner, 1987).

Analogy-making as modeled in the Copycat program is in agreement with several aspects of Gentner's theory. We agree with the main idea of systematicity: that in general, the essence of a situation—the part that should be mapped—is a high-level coherent whole, not a collection of isolated low-level similarities. In Copycat, the pressure toward systematicity is an emergent result of several pressures:

- The pressure, coming from codelets, to perceive relations and groupings within strings.
- The pressure to see things abstractly (which itself emerges from the preference for using descriptions of greater conceptual depth, and from the tendency of deeper concepts to stay active longer).
- The pressure to describe the change from the initial to the modified string in terms of relationships and roles, since these tend to be deeper than attributes (e.g., in formulating a rule for the change **abc** ⇒ **abd**, it is in general better to describe the **d** as "the successor of the rightmost letter" rather than as "an instance of *D*").
- The greater salience of larger relational structures (e.g., a whole-string group), which makes them more likely to be paid attention to, and hence mapped.
- The high strength of correspondences between large relational structures (such as whole-string groups): such correspondences are strong not only because they involve large structures, but also because they are based on many concept-mappings.
- The pressure toward forming a set of compatible correspondences that, taken together, form a coherent worldview.

Gentner captures some important points in her characterization of a "good" analogy, and the same pressures exist in Copycat: her pressure toward "clarity" is enforced by our prohibition of many-to-one or one-to-many mappings without first making the "many" into a grouped unit; her pressure toward "richness" corresponds to Copycat's preference for having many correspondences and many concept-mappings underlying a correspondence; and the program's drives toward abstraction and systematicity are described above. But note that Gentner's definition of "abstraction" (order of a relation) is not the same as the notion in Copycat of "conceptual depth" (which was described in Section 3.2). In Copycat, there is no logic-based definition for conceptual depth, but rather these values are assigned by hand,

with quite high values sometimes going to concepts that Gentner might call "attributes" (such as *first*, which could be seen as an attribute of an **a**).

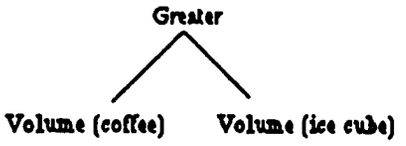
Although there are points of agreement, there are also some fundamental issues on which our approach and Gentner's disagree, and some of the most important aspects of analogy-making addressed in the Copycat project are not dealt with in Gentner's theory and model.

Genter's abstractness and systematicity principles capture something important about analogy-making, but there are often other pressures in an analogy: both superficial and abstract similarities that may not be parts of systematic wholes, but are still strong contenders in a competition. An example of this in the Copycat domain is Variant 9 from Chapter 5:

aabc ⇒ **aabd**
ijkk ⇒ ?

The abstractness and systematicity principles would, I think, argue for the answer **ijll**, since the attribute *sameness-group* describing the group of **a**'s and the group of **k**'s is firstly merely an attribute, and secondly is not related to the systematic set of successor relations in each string; according to the systematicity principle, it should thus not be mapped, but should be ignored. However, many people feel that the two groups should map onto each other nonetheless, and that the best answer is **hjkk**, in spite of what I think would be an *a priori* dismissal by the structure-mapping theory. Making any analogy involves a competition between rival views, and one cannot be certain ahead of time that the mapping with the highest degree of systematicity (in Gentner's sense) will be the most appealing.

Another problem with Gentner's theory is that for any complex situation, there are many possible sets of relations that exhibit systematicity, and it is not explained how certain ones are considered for mapping and not others, on syntactic grounds alone. For example, suppose the heat-flow domain had contained the following relation:



There would be no reason, based on syntax alone, to prefer the structure concerning temperature over this structure for mapping; if this structure were chosen, the analogy-maker would mistakenly learn that, just as the pressure differential causes the water flow, the

volume differential causes the heat flow. There is not even a semantic connection given between *temperature* and *heat* that might guide one to suspect the implication of temperature in understanding how heat flow occurs. In short, which facts are part of a relevant systematic whole, and which are isolated and irrelevant, depends on the situations at hand and cannot be determined by syntactic structure alone.

By contrast, in Copycat, the mechanisms for deciding what things to concentrate on and which mappings to make involve semantics: they involve activation of concepts in the Slipnet in response to perception of instances of those concepts (or of associated concepts) in the letter strings, competition among objects clamoring to be noticed and among various descriptions of objects and relationships between objects, and certain *a priori* notions of salience. For the Structure-Mapping Engine, not only are the attributes and relations in each situation laid out in advance, but there is no notion of differential relevance among them: which ones get used in an analogy is entirely a function of the syntactic structure connecting them. In Copycat, the notions of differential relevance and non-black-or-white inclusion of concepts in a situation (via probabilities as a function of differential activation in the Slipnet)—and of the program *itself* bringing in the concepts to be used to describe the situation—are fundamental, since Copycat is a model of how situations are interpreted as well as how mappings are made between them, and of how the two processes interact. The philosophy of Gentner and her colleagues is that the interpretation stage and the mapping stage can be modeled independently; that there are, in effect, separate "modules" for each. In contrast, a philosophy underlying the Copycat project is that the two are inextricably intertwined; the way in which the two situations are understood is affected by how they are mapped on to each other, as well as vice versa. Such an interaction could be seen in the screen dumps given in Chapter 4. For example, in "**abc** ⇒ **abd**, **kji** ⇒ ?", how **abc** was mapped to **kji** had a profound influence on how the latter was interpreted, and vice versa. This issue of the necessity of integrating these two processes is discussed further in Chalmers, French, and Hofstadter (1990).

Another fundamental difference between our approach and Gentner's is that her theory does not include any notion of conceptual similarity or of slippage, notions absolutely central to the Copycat project. In the water-flow–heat-flow example given above, the representations of the two situations are sufficiently abstract to make the analogy a virtual isomorphism. For example, the concepts of *water flow* and *heat flow* have both been abstracted in advance into a general notion of *flow*. Likewise, in another analogy that Gentner

describes, in which the hydrogen atom is mapped onto the solar system, all the important predicates in both situations have the same labels (e.g., *attracts*, *revolves around*, *mass*). This is necessary because of the theory's reliance on syntax alone. If this "identicality" constraint were to be relaxed, semantics and context-dependence (i.e., some knowledge of conceptual proximity and how it is affected by context) would have to be brought in. But at present, since the concepts contained in the preconstructed representations are always in a sufficiently abstract form, there is no need for a Slipnet-like structure in which various concepts flexibly become more or less similar to one another in response to context. The analogy is already effectively given in the representations.

Another problem with Gentner's theory is that it relies on a precise and unambiguous representation of situations in the language of predicate logic. The structure-mapping theory's reliance on syntax alone requires that situations be broken up very clearly into objects, attributes, functions, first-order relations, second-order relations, and so on. For example, the water-flow–heat-flow analogy includes the following correspondences:

water ⇒ *heat* (both are objects);

coffee ⇒ *beaker* (both are objects);

flow (beaker, vial, water, pipe)

⇒ *flow (coffee, ice cube, heat, bar)* (both are 4-place relations);

But suppose that, in the heat-flow situation, *heat* had been described not as an object, but as an attribute of coffee, as in *emits-heat (coffee)*, or that *flow* had been given as a 3-place rather than as a 4-place relation: *flow (coffee, ice cube, heat)* where the means of heat flow is considered to be irrelevant, or suppose that, in the water-flow situation, *water flow* had been given as a 5-place relation: *flow (beaker, vial, water, pipe, 10 cc per second)* where the rate of flow is included. Any of these quite plausible changes would prevent a successful application of the structure-mapping theory. The problem is that in the real world, the categories "object", "attribute", and "relation" are very blurry, and people (if they assign such categories at all) have to use them very flexibly, allowing initial classifications to slide if necessary at the drop of a hat. And to do this, semantics must be taken into account (this point is also made by Johnson-Laird, 1989). In the water-flow–heat-flow representation, *heat* is presented as an *object*, but in the solar-system–atom representations it could plausibly be given as an *attribute* of the sun (e.g., *generates-heat (sun)*). The classification of *heat* as an object is necessary for the water-flow–heat-flow analogy to work, but is not necessarily a classification that the analogy-maker would make *before* figuring out what the mappings

were. It seems likely that any two people (or even one person, at different times) would produce very different predicate-logic representations of, say, the water-flow situation, no doubt differing on which things were considered to be objects, which were attributes, which were relations, how many arguments a given relation has, and so on. Thus, a serious weakness of the structure-mapping theory is its inability to deal with any flexibility in the representation of situations.

To be sure, Copycat also breaks up a situation's representation too cleanly into object-attributes (descriptions) and relations between objects, where many people would not do so. For example, in the string **aaabcd**, should the fact that the **b** is the alphabetic successor of the group of **a**'s be represented as a relation *between* the two objects, or as a description belonging exclusively to the **b**? It depends on the context. If the problem were

aaabcd ⇒ **aaaxcd**
pqqqrs ⇒ ?,

then one could plausibly use that fact as a description, viewing the **b** and the **r** as corresponding because they are both "successor of the sameness group", and answer **pqqqxs**. However, such a description might not be applied to the **b** in **aaabcd** if the problem were

aaabcd ⇒ **aaabce**
pqqqrs ⇒ ?

In the latter, to get the answer **pqqqrt**, the **a**–**b** successor relation would be used only as one of the relations tying together the initial string. Copycat is currently unable to make descriptions such as "successor of the sameness group", but I believe that the architecture of Copycat would allow one to fairly straightforwardly give it the ability to make and use such descriptions appropriately. The possibility for such real-time representational flexibility is lacking in a program like SME, which relies solely on the syntax of predicate-logic representations that are supplied to it before the fact. For such a program to work, the representations have to be tailored carefully.

Thus, both the architecture and purpose of the Structure-Mapping Engine are quite different in spirit from those of Copycat. Although SME is meant to simulate human analogy-making, in that it models which types of structures tend to be mapped from one situation to another, and which of the various possible mappings will be preferred, it doesn't attempt to model concepts or perceptual processes in the way Copycat does, and the exhaustive search it performs through all consistent mappings is not meant to be psychologically

plausible. Rather, it seems that SME is meant to be an automatic way of finding what the structure-mapping theory would consider to be the best mapping between two given representations, and of rating various mappings according to the structure-mapping theory, which ratings can then be compared with those given by people.

In summary, Copycat has a store of knowledge about letter-strings that is structured independently of any particular problem, and that is adapted *by the program* to each new problem. SME has no permanent store of knowledge; information about each new situation is put into predicate-logic notation *by people* only after a problem is given, and for each new problem, a new set of facts specific to the problem is needed. SME also relies on rigid predicate-logic descriptions discussed above, where the representations are fixed at the start of processing and cannot be altered by the program. Copycat starts out with raw, unperceived situations, and it is in the process of describing these situations and their relations to one another that the concept network (the Slipnet) is modified dynamically and eventually settles into a certain pattern of activations and conceptual proximities. It is impossible to know ahead of time which concepts will be important and what reformulations and slippages will need to take place in the course of making an analogy. SME does not address these issues, but rather starts out with already-formed representations of situations, with the task of deciding which mappings are preferable. The structure-mapping theory makes some very useful points about what features appealing analogies tend to have, but in dealing only with the mapping process while leaving aside the problem of how situations become understood and how this process of interpretation interacts with the mapping process, it leaves out some of the most important aspects of how analogies are made.

8.1.2 Holyoak and Thagard

Keith Holyoak and Paul Thagard have built a computer model of analogical mapping (Holyoak & Thagard, 1989), based in part on theoretical and experimental work by Holyoak and his colleagues (Gick & Holyoak, 1983; Holland, Holyoak, Nisbett, and Thagard, 1986), and inspired in part by research by Marr and Poggio on constraint-satisfaction networks used to model stereoscopic vision. The computer model, ACME (Analogical Constraint Mapping Engine), is similar to SME in that it uses representations of a source situation and target situation given in sentences of predicate logic, and makes an analogical mapping consisting of pairs of constants and predicates from the representations. In fact, ACME has been tested on several of the same predicate-logic representations of situations that SME

was given, including the water-flow and heat-flow representations. For ACME, a mapping between two situations is based on the following five constraints:

- Logical compatibility: A mapped pair has to consist of two elements of the same logical type. That is, constants are mapped onto constants and n-place predicates are mapped onto n-place predicates. For example, in the water-flow–heat-flow analogy, *water* could map on to *water*, but not onto *flow*, because the former is a constant and the latter is a 4-place relation.

- Uniqueness: Each source element must map onto at most one target element.

- Relational consistency: The various pairings making up a global mapping must support each other. For example, if *flow* in one situation maps onto *flow* in the other, then that supports a mapping between *water* and *heat*, since they play corresponding roles in the *flow* relations.

- Semantic similarity: Pairings of predicates whose elements have similar meaning are preferred.

- Role identity: This constraint applies to analogies between problem-solving situations, which are represented in terms of initial states, goal states, solution constraints, and operators. This constraint requires that initial states map to initial states, goal states to goal states, and so on.

The model takes as input a set of predicate-logic sentences containing information about the source and target domains (e.g., water flow and heat flow), and it constructs a network of nodes, where each node represents a syntactically allowable pairing between one source element and one target element (a constant or a predicate). (Here, "syntactically allowable" means adhering to the logical-compatibility constraint.) A node is made for *every* such allowable pairing. For example, one node might represent the *water* ⇒ *heat* mapping, whereas another node might represent the *water* ⇒ *coffee* mapping. Links between nodes in the network represent constraints; a link is weighted positively if it represents mutual support of two pairings (e.g., there would be such a link between the *flow* ⇒ *flow* node and the *water* ⇒ *heat* node, since *water* and *heat* are counterparts in the argument lists of the two *flow* relations), and negatively if it represents mutual disconfirmation (e.g., there would be such a link between the *flow* ⇒ *flow* node and the *water* ⇒ *coffee* node). The network also has a "semantic unit": a node that has links to all nodes representing pairs

of predicates. These links are weighted positively in proportion to the "prior assessment of semantic similarity" (i.e., assessed by the person constructing the representations) between the two predicates. In addition, it has a "pragmatic unit": a node that has positively weighted links to all nodes involving elements (e.g., *water*) deemed ahead of time (again by the person constructing the representations) to be "important". Once the network is in place, a spreading-activation relaxation algorithm is run on it, which eventually settles into a final state with a particular set of activated nodes representing the winning matches.

There are several points of agreement between the philosophy of this model and that of the Copycat program. We share the idea that analogy-making is closely related to perception and should be modeled with techniques inspired by models of perception. We also share the belief that analogies emerge out of a competition among pressures (or "soft constraints"), involving a large number of local decisions that give rise to a larger coherent structuring. And we agree that the pressure toward systematicity (as described by Gentner) emerges from other pressures. Copycat has counterparts to Holyoak and Thagard's relational-consistency constraint (Copycat's pressure toward compatible correspondences) and their semantic-similarity constraint (in Copycat, correspondences involving close concept-mappings are strong).

There are, however, deep differences between Copycat and ACME, related to Copycat's differences with SME discussed in the previous section. First, like SME, ACME tries *all* syntactically plausible pairings, a method that is both computationally infeasible and psychologically implausible in any realistic situation. For example, in making a Watergate–Contragate analogy, do we consider a mapping between Nixon and every person involved in Contragate, including Fawn Hall, Daniel Inouye, Ed Meese, and Dan Rather? Or even less plausibly, do we consider mapping Gerald Ford to the Contras' base camp in Honduras, or to the chair Oliver North sat in while testifying before Congress? Yet these are all plausible, according to the logical-compatibility constraint, in which semantics plays no role at all. The existence of this exhaustive (though parallel) search through all possible mappings shows that ACME is not attempting to model how people search through such possibilities, whereas this is one of the Copycat project's main focuses. In Copycat, although any initial-string object can *in principle* be compared with any target-string object, an exhaustive search is avoided thanks to the parallel terraced scan, in which comparisons, if they are made at all, are made at different speeds and to different levels of depth, depending on estimates of their promise.

A major problem in the ACME system is the same problem I discussed with respect to SME: the representations of knowledge used are rigid, and are also tailored specially for each new analogy. ACME uses the same representation as did SME for the water-flow–heat-flow analogy, so the same issues discussed in the section on Gentner et al. apply here. Again, the program has no ability to restructure its descriptions or to add new descriptions in the course of making an analogy; the descriptions are constructed by a person ahead of time and are frozen. ACME differs from SME in that it has a "semantic unit" giving semantic similarities, which correspond in some sense to those embodied in Copycat's Slipnet, but the similarities are also decided in advance by the programmer for the purposes of the given analogy, and are frozen. Unlike Gentner et al., Holyoak and Thagard recognize the necessity of considering semantics as well as syntax, but the problem is that it is impossible in general to have a "prior assessment of similarities" (as encoded in ACME's semantic unit); rather, analogy-making is all about similarities being reassessed in response to pressures that weren't apparent ahead of time.

ACME also leaves aside the question of how concepts come to be seen as important in response to pressures; this is taken care of by the pragmatic unit, which encodes the programmer's prior assessment of what is important in the given situations. The pragmatic unit could be said to correspond to the activation of Slipnet nodes and to the *importance* values of objects in Copycat. But again, unlike in Copycat, where these values emerge in response to what the program perceives, in ACME, the pragmatic unit is set up by a person and then frozen for each new problem. Thus, like SME, ACME does not deal with another of Copycat's main focuses: how concepts *adapt* to different situations. ACME, like SME, models only the "mapping stage" of analogy-making, but, as was said before, a philosophy underlying Copycat is that the mapping process cannot be separated from the processes of perceiving and reformulating perceptions and assessments of similarities in response to pressures. Holyoak and Thagard (1989) themselves point out that their model does not address this issue—they call it the issue of "re-representation"—and acknowledge that it will often be necessary to interleave mapping with manipulation of the representations, taking into account top-down pressures—which is essentially just what Copycat does.

Since ACME's knowledge is set up ahead of time, the program's success, like that of SME, is totally dependent on the representations it is given. In the examples given by Holyoak and Thagard (1989), the representations of the source and target matched each other almost perfectly; the essence had been distilled in exactly the right form for making

an analogy. Thus the program was quite successful, even though experiments done by Gick and Holyoak (1983) showed that many people have a hard time with the same analogies. As with SME, it is very doubtful that the representations given to ACME could have been made by someone who didn't already have the mapping in mind, and it is almost certain that the program would not succeed if the representations were made independently by two different people.

8.1.3 How Real Are These "Real World" Analogies?

One of the criticisms that has been made of Copycat (as well as of Evans' program, to be discussed in the next section) is that it makes analogies in an idealized microworld, whereas other analogy-making programs work in more complex, real-world domains. On the surface it would seem that SME and ACME make real-world analogies that are much more complex than the "toy" problems Copycat deals with. But if one looks below the surface (as I did here for the water-flow–heat-flow example), it can be clearly seen that the knowledge possessed by these programs (that is, the knowledge given to them for each new problem, in the form of sentences of predicate logic), in spite of the real-world aura of words like "pressure" and "heat-flow", is even more impoverished than Copycat's knowledge of its letter-string microworld. The programs know virtually nothing about concepts such as *heat* and *water*—much less than Copycat knows about, say, the concept *successor group*, which is embedded in a network and can be recognized and used in an appropriate way in a large variety of diverse situations. For example, **abc**, **aabbcc**, **cba**, **abbbc**, **mrrjjj**, **mmrrrjjjj**, **jjjrrm**, **abbccc**, **xpqefg**, and **k** (a single-letter successor group) can all be recognized as instances of successor groups, given the appropriate pressures.[1] This is not the case for, say, SME's and ACME's notion of "heat" as given for the purpose of making a water-flow–heat-flow analogy. There the notion of "heat" has essentially no semantic content and certainly cannot be adapted to any other situation. Nor can these programs recognize heat or a heat-

[1] Myriad other examples of successor groups, with different degrees of abstruseness, can be formed in the letter-string domain. Many are beyond Copycat's current recognition capabilities, though the same perceptual mechanisms the program has now could, I believe, be extended fairly readily to recognize more complex instances, such as **ace** (a "double-successor" group), **aababc** (which can be seen as a "coded" version of **abc** when parsed as **a-ab-abc**), **kmxxrreeejjj** (which could be described as "11-22-33"), **axbxcx** (where the **x**'s form a ground for the figure **abc**), **abcbcdcde** (which could be parsed **abc-bcd-cde**), and so on.

like phenomenon. These programs are purported to make analogies involving the concepts *heat* and *water*, but the programs have absolutely no sense of "heat" or "water" themselves as *categories* and cannot make the very analogies required to recognize instances of these categories (as humans do) in a variety of contexts.

Thus the claim that Copycat's microworld is a "toy domain" while these other programs are solving real-world problems is truly unfounded, and is based on a tendency of people to attribute much more intelligence to a program than it deserves based on real-world-sounding words it uses (such as "heat")—concepts that are extremely rich for people, but are almost completely empty as far as the program is concerned. (McDermott, in his article "Artificial Intelligence Meets Natural Stupidity" (1981) writes humorously but incisively about some related problems in artificial-intelligence research methodology.) Programs that use words with real-world connotations but that are nonetheless completely devoid of semantic content as far as the program is concerned have great potential to be misleading. An "all the cards are on the table" quality is one of the advantages of using explicit microworlds for research in artificial intelligence.

8.1.4 Evans

Thomas Evans' ANALOGY program (Evans, 1968) was written in the 1960's to solve IQ-test-like geometric-analogy problems (many of which were taken from actual examinations given to college-bound high-school students by the American Council on Education). A sample problem is given in Figure 8.3. The idea is to choose the box in the bottom row that has the "same" relation to box C as box A has to box B. ANALOGY is given as input the information that box A contains two simple closed curves and one dot, along with the coordinates of the vertices and the curvature of the lines; similar information is given for all the other boxes. The program then computes, for each box, properties of the figures inside it and relations among them, using a predetermined set of possible properties and relations and a "substantial repertoire of 'analytic geometry' routines". For example, for box A in

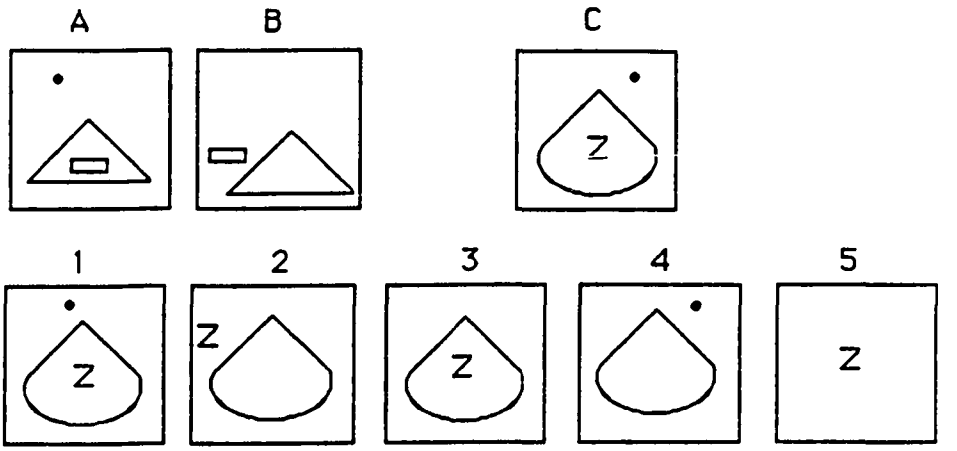


Figure 8.3: A sample problem from Evans' geometric-analogy domain.

the problem shown, the program would find the following relations:

(INSIDE rectangle1 triangle1),
(ABOVE dot1 triangle1),
(ABOVE dot1 rectangle1),
(LEFT dot1 triangle1),
(LEFT dot1 rectangle1).

(Note: I use words like "rectangle" and "triangle" only for clarity; the program does not have the concepts *triangle* or *rectangle*, and has no notion of similarity at the conceptual level between, say, two different triangles. It was not able, therefore, to solve problems involving rules such as "Replace all triangles by squares".) In order to describe the change from box A to box B, the program uses a given set of possible transformations to make all possible mappings from the figures in box A to those in box B. The repertoire of possible transformations contains: removal of objects, addition of objects, rotation of objects, uniform scale-change of objects, and horizontal and vertical reflection of objects. From this set of mappings the program creates a set of rules describing the change from A to B.

Next, the program tries to match box C with each of the numbered answer boxes, discarding an answer box if the matching does not agree with the A-to-B rules in terms of the number of objects added, removed, or matched. In the example, answers 1 and 5 are discarded for this reason. The program then does a (potentially huge) exhaustive search through all possible ways of mapping C to each of the remaining answers, given the possible

A-to-B rules that have been formed. In this way, a set of possible C-to-answer rules is constructed (each one is a weakened form of one of the A-to-B rules, from which statements that are not true of the C-to-answer match are removed). Each of these C-to-answer rules is scored using a complicated procedure that values the amount of information the rule contains (roughly, the length of the rule); this reflects the heuristic that strong C-to-answer rules are ones requiring little alteration of the original A-to-B rule. The answer given by the rule with the highest score is chosen.

Evans' geometric analogies are very much in the spirit of Copycat's microworld, not merely because the analogies are in the form of "proportions", but because they are abstract; although such analogies have no conscious "purpose" (as in problem-solving), humans have definite feelings about what makes for a deep mapping and what makes for a shallow one. The fact that such abstract analogy problems are used without argument on intelligence tests (as at least requiring some aspect of intelligence to solve them) shows how generally accepted is the point I made earlier: that people are able to bring to bear their perceptual and analogical abilities in an idealized domain; indeed, they are unable *not* to. This domain, like Copycat's, has the potential for very interesting and creative analogies, in spite of its limited number of concepts. Evans' domain is closely related to the extraordinarily rich domain of Bongard problems (Bongard, 1970), which was one of the early inspirations for the Copycat project.

Although Evans' domain is potentially very rich, his program was able to solve only a very limited set of problems in this domain. For example, the transformations from box A to box B are restricted to those involving the addition and removal of objects, and Euclidean transformations (rotation, reflection, uniform-scale change). Therefore, the program would not be able to deal with a problem in which a triangle in box A was transformed into a square in box B, even if they both played the same role (say, "the object containing the Z"); there is no notion of conceptual similarity or of similarity of roles. The program also has no notion of grouping; thus, it would not be able to solve the problem given in Figure 8.4. The program would be stymied by the fact that the number of dots in A is different from the number of dots in C. The program is able to deal only with problems in which the number of parts added, removed, and matched in the A-to-B transformation is the same as in the C-to-answer transformation. All the problems that Evans' program attempted (he displays the entire set of 20 problems the program was tried on) had the same number of objects in A and C. In Copycat, the kinds of similarities possible between the initial and

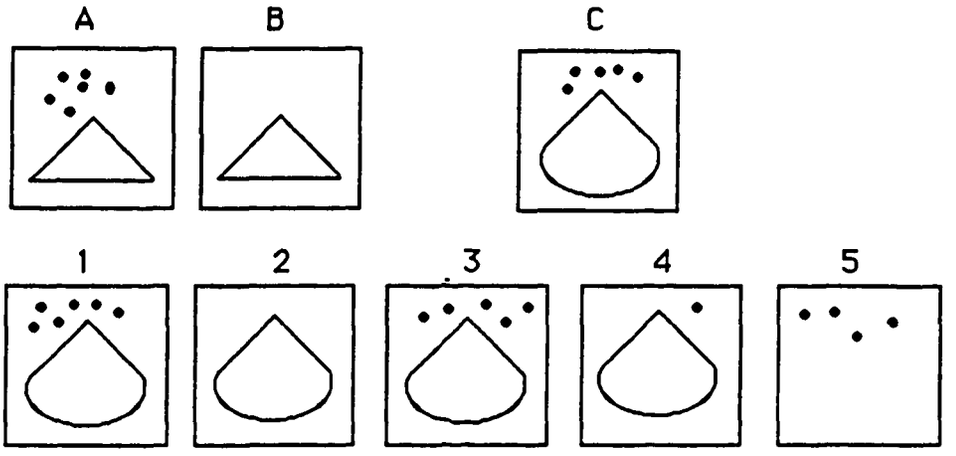
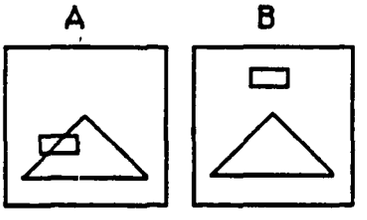


Figure 8.4: A problem that requires grouping, and that Evans' program would not be able to solve.

target strings can be much more complex. In addition, each of Evans' problems had exactly one strong answer, whereas many of Copycat's problems have more than one good answer. Such problems are among the most interesting, because they bring out very clearly issues of how various pressures compete.

ANALOGY is nonetheless more similar in many ways to Copycat than are the other analogy-making programs described in this section. As in Copycat, in Evans' system the situations given to the program have only minimal descriptions attached, and the program itself has to perceive the relations among the various parts. The program also has a notion of adapting the A–to–B rule to fit the matchings between C and the various answers, which is roughly similar to rule translation in Copycat. In addition, in Evans' program, context exerts top-down pressure on the way things are perceived; for example, given the following A and B boxes,



the program will decompose the figure in box A into a rectangle and a triangle, since these are the objects in box B (though since the program has no *concept* of "rectangle" or "trian-

gle", it would not be able to perceive the similarity between A and B if the corresponding figures happened to be of slightly different shapes). However, the role of context is limited; for example, the program's perception of box C has no effect on the perception of box A. In Copycat, such contextual effects can be very important (as they were in the problem "**abc ⇒ abd, xyz ⇒ ?**", where the **a** is described as *first* in response to what is perceived in **xyz**). In general, the processes of description, mapping, and rule formation in ANALOGY do not interact with each other as they do in Copycat. Evans saw the desirability of some kind of interaction among the various processes, but did not implement it; his program proceeds in stages in a strict serial fashion, and no backtracking for restructuring of perceptions is done.

Aside from the similarities mentioned above, the workings of Evans' program are very different from Copycat's. ANALOGY has nothing like a Slipnet; there is no notion of *conceptual* similarity, only a rigid notion of *geometric* similarity. This, along with the fact that the roles between box A and box C have to be identical, severely limits the kinds of problems that ANALOGY is able to solve. Evans recognized that it is not always possible to adapt an A-to-B rule to a C-to-answer rule by weakening it; in some cases, translation with slippage is needed. In fact, he gave one example of a problem where this was needed, but the great majority of his problems involved only identity concept-mappings, so he was not very concerned by this issue. He did implement a very rough kind of slippage, in which one word in the A-to-B rule (e.g. ABOVE) is replaced by another word in the C–to–answer rule (e.g. LEFT). This is done only if, at the last stage, there isn't one answer that is clearly stronger than the others. Then the program goes back to the A-to-B rule-building stage and generates some "variant rules", using this substitution technique. Evans does not explain exactly how this was done.

Another major difference is the lack in Evans' program of anything like a parallel terraced scan. Instead, his program adopts the brute-force method of making all possible relations, transformations, and rules, and then scoring them. This method has the usual problems of psychological implausibility and combinatorial explosion (though Evans cannot be faulted on the psychological implausibility, because his program was meant to be an AI program, not a cognitive model). ANALOGY was tried only on cases where there was no ambiguity and little competition, so there were only a small number of possibilities for the program to consider in each case. It would be impossible to use this method on more complex problems with more facets.

In summary, ANALOGY was an interesting early attempt at mechanizing an aspect of intelligence, though it used a brute-force approach with a completely deterministic control structure that proceeded through stages in a fixed manner. It was not meant to be a cognitive model of concepts or perception. Its performance is in some ways impressive, since it was able to solve a number of problems considered hard enough to put on a college-entrance test, but as has been pointed out, the range of problems it could solve was actually quite limited.

8.2 Comparisons With Related Artificial-Intelligence Architectures

8.2.1 Seek-Whence

In addition to Jumbo, another precursor to Copycat was the Seek-Whence project. Hofstadter designed the domain and the original ideas for the architecture (Hofstadter, Clossman, & Meredith, 1982), which was based on the architecture of Jumbo, described in Chapter 3. The Seek-Whence program was developed by Marsha Meredith (1986). Seek-Whence is a discovery-and-extrapolation program; it tries to find the underlying regularity of a sequence of integers—in other words, to "seek whence" the sequence comes. The sequences it works on have patterns rather than mathematical functions underlying them, in which the major organizing concepts are successorship, predecessorship, sameness, and symmetry. The following are some sample sequences given to the Seek-Whence program:

1 2 3 4 5 6 ...;
1 1 2 2 3 3 ...;
1 8 5 1 8 5 ...;
1 1 2 1 2 3 1 2 3 4 ...;
2 1 2 2 2 2 2 3 2 2 4 2 ...

Seek-Whence is given the terms of a sequence one by one, and it tries as soon as it can to propose a hypothesis to explain the sequence. It is thus often required to reformulate its hypothesis in light of new, contradicting evidence (new terms).

Sequence-extrapolation programs in artificial intelligence (e.g., Pivar & Finkelstein, 1964) have typically dealt with *mathematical* sequences, such as "1 2 4 8 16 ...", or "1 2 5 15 42 ..." (whose second differences are every third prime), and have approached them by trying out possible solutions from a standard repertoire of mathematical knowledge and tricks (e.g., primes, powers of two, Fibonnacci numbers), often recursively applying the

techniques to derived sequences formed by taking every other term, every third term, first differences, etc. This is very different from the goal of the Seek-Whence project, which is to model a much more general sort of pattern-spotting ability: sequences in the Seek-Whence domain contain the essence of many central issues of pattern-recognition in general (Hofstadter, Clossman, & Meredith, 1982). (The sequences Meredith's program was able to tackle were more varied and general than the cyclical, fixed-length period sequences dealt with by a well-known program written by Simon & Kotovsky, 1963, which will be discussed further in the next section.) Analogy-making plays an essential part in solving these sequences; for example, to find a coherent interpretation for the sequence "1 2 1 1 3 1 1 4 1 ...", one must map hypothesized segments against each other, perceiving corresponding *roles* within segments (for example, a reasonable parsing is "121-131-141 ...", with the role played by the "2" in 1 2 1 corresponding to the role played by the "3" in 1 3 1, and so on). What originally gave rise to the Copycat project was Hofstadter's desire to further isolate this essential role of analogy-making in Seek-Whence.

Like Jumbo, Seek-Whence has a nondeterministic parallel architecture involving codelets, and the program is based on many of the ideas Hofstadter first developed in the Jumbo project. As in Jumbo, a major part of the operation of Seek-Whence is the construction, destruction, and reformulation of groupings built out of the raw numerical data of the sequence (e.g., the program could parse the sequence "1 1 2 2 3 3 ..." into groups: "11-22-33 ..."). Seek-Whence proceeds by building such groupings, using the groupings to construct a hypothesis enabling it to predict the next number in the sequence, and reformulating that hypothesis via slippage when an unexpected piece of new evidence (i.e., an unexpected new term in the sequence) requires such action. Such reformulation sometimes requires modifying or destroying the groupings the program has already made.

The Copycat and Seek-Whence projects deal with many of the same issues, and thus there are many correspondences between the Seek-Whence and Copycat programs; some of my ideas for Copycat have come from Meredith's solutions to various implementation problems. Copycat further develops many of the mechanisms used in Seek-Whence (as it did with Jumbo) and includes many mechanisms lacked by the Seek-Whence program, so there also are many major differences between the two programs.

Much of the architecture of Seek-Whence is admittedly "ad hoc"; for example, the program uses a large number of special-purpose domain-specific codelets and structures. In Copycat, I tried to avoid this problem by making codelets and perceptual structures as

general and domain-independent as possible. Copycat's conceptual system is much richer than that of the current version of Seek-Whence; the latter's Slipnet has fewer nodes and no named relations, only undifferentiated "slipping links", with no notion of activation or conceptual distance—unlike in Copycat, the dynamics of the Slipnet was not a central part of the Seek-Whence model. In Seek-Whence, slippage does not occur unless something goes wrong; it is not nearly as central a focus in the Seek-Whence program as it is in Copycat.

Another extremely important issue for both Copycat and Seek-Whence—how top-down pressures work to influence the program's conceptualization of the problem at hand—was dealt with by Meredith in only a limited way; the Copycat project has made considerable progress on this issue. For example, Seek-Whence could not solve the sequence "1 2 2 3 3 3 4 4 4 4 ...", because of its lack of responsiveness to emerging top-down pressures. Since it was given the sequence one term at a time, the first two terms—1 and 2—put the program on the track of successorship and successor groups, and it could never recover enough to perceive the sequence's sameness groups. One problem is that the program clings too tenaciously to its first organizing notion, and another problem is that it lacks the kinds of top-down codelets that Copycat has, such as codelets that expressly look for sameness groups if several sameness relations have been spotted. (Interestingly, Seek-Whence could solve "2 2 3 3 3 4 4 4 ...", since that sequence allows it to start off on the right foot.) The program lacked much of the interaction between bottom-up and top-down pressures that is an essential part of Copycat, as well as many other architectural features that are present in Copycat, such as dynamically varying activation and link-lengths in the Slipnet, different degrees of conceptual depth for different nodes, and temperature, among others.

8.2.2 Simon and Kotovsky

Simon and Kotovsky's work on pattern perception and sequence extrapolation (1963; also Simon, 1972, and Kotovsky & Simon, 1973) involves a domain similar in some ways to those of Seek-Whence and Copycat, though the approach is completely different. Meredith (1986) gives a discussion of Simon and Kotovsky's work with respect to Seek-Whence, and much of what she says also applies to a comparison with Copycat. Simon and Kotovsky studied human performance on understanding and extrapolating letter sequences such as:

c d c d c d ...;

q x a p x b q x a ...; and

r s c d s t d e t u e f ....

(The last consists of two interleaved sequences.) Simon and Kotovsky's goal was to show that people build a symbolic mental model of a given sequence based on a set of descriptions such as "successor", "predecessor", and "sameness", and that they use this model (or *rule*) to extrapolate the sequence. Part of the project was the construction of a computer program to model this process. There were actually two programs: one for producing a sequence, given a pattern description, and, more interesting, one for coming up with a pattern description, given a sequence. The latter program first looked for two possible types of patterns in the sequence: (1) periodicity (e.g., "c d c d c d ...", where the same symbol occurs in every second position, or "d e f g e f g h f g h i ...", where the next symbol occurs at every fourth position) or (2) a relation that is interrupted at regular intervals by another relation (e.g., "a a b b c c d d ...", where sameness relations are interrupted every second position by a successor relation). (All the sequences were cyclic, with fixed-length cycles.) Once such a pattern was discovered, sameness, successor, and predecessor relations were explored between the successive terms within a period, or between terms in corresponding positions of successive periods.

Several variants of the program were written, with different degrees of success. The program (or at least some version of it) agreed with people on which sequences were hard (as a function of which ones it could solve and how long it took).

Simon and Kotovsky's program was not a model of concepts or perceptual processes in the same way Copycat is. It searched through possible ways of describing a given sequence in a deterministic and exhaustive manner, trying out all the possibilities in its repertoire until one of them worked. There was no notion of top-down–bottom-up interaction and competition, no change in processing as a result of what had already been discovered, no notion of a parallel terraced scan, and no notion of fluid and adaptable concepts (rather, it had fixed concepts of successor, predecessor, sameness, and periodicity). Also, although the program worked in a domain that is similar in some ways to that of Copycat, Simon and Kotovsky were specifically studying *sequence* perception, whereas Copycat is not, at a deep level, about linear sequences; its focus is much broader. Thus, though the Copycat project shares certain general goals and methodology with Simon and Kotovsky's work (i.e., investigating the mechanisms of pattern recognition by studying it in an idealized, abstract domain), *which* aspects of perception are being studied and *how* perception is modeled are quite different for the two projects.

8.2.3 Hearsay-II

As was mentioned earlier, many of the ideas for Copycat's architecture were originally inspired by the Hearsay-II speech-understanding system (Erman et al., 1980). The input to Hearsay-II is a waveform generated by a spoken utterance, and Hearsay-II interprets it through the cooperation of various "knowledge sources", each of which is able to perform a specific task, such as dividing the waveform into segments, creating phones and phonemes from segments, creating syllable-class hypotheses from phonemes, creating word hypotheses from syllables, and so on. The knowledge sources build data structures—representing various levels of interpretation of the utterance—on a global blackboard, which is the locus of communication among the various knowledge sources. The need for diverse knowledge sources to deal with different levels of description reflects the diversity of processes needed in perception. This is the intuition behind the various types of codelets in Copycat. But Copycat's codelets are somewhat different from Hearsay-II's knowledge sources: codelets perform small, very local tasks, whereas a knowledge source deals with all the data at its level of abstraction (e.g., one knowledge source segments the entire waveform). As was discussed in previous chapters, the idea in Copycat is that early on in a run, the program does not have enough information to make large-scale *intelligent* decisions about which structures to build. Instead, structure-building is accomplished via a large number of small, local decisions that allow many different possibilities to be scouted out and then looked at more deeply if more consideration seems warranted. Later in a run, when more structures have been built and temperature is low, codelets tend to act more like Hearsay-II knowledge sources, building global structures (e.g., a successor group comprising an entire string). The path for such a global structure has been laid both by the codelets that build the underlying structures that support it, as well as by the codelets that scouted out (and perhaps rejected or slowed down consideration of) alternative pathways.

The various knowledge sources in Hearsay-II, in the process of moving from a raw waveform to a fully parsed utterance, build many levels of data structures, where each level is built on the basis of hypotheses built at the immediate lower level (segments make up phones, phones make up phonemes, phonemes make up syllables, and so on). This is similar to the way codelet chains in Copycat build various levels of perceptual structures, starting with three raw letter-strings, and ending with high-level descriptions and a coherent mapping. As in Copycat, an important part of Hearsay-II's architecture is an interaction between top-down and bottom-up processing, where structures built at lower levels provide

evidence for higher-level hypotheses, and vice versa.

In Hearsay-II, a knowledge source becomes activated through demons with various levels of conditions, preconditions, and pre-preconditions. This is roughly similar to the various stages of exploration and evaluation that take place in Copycat before a given structure is built. But although both programs perform a parallel exploration of different pathways, and both have competition between different interpretation hypotheses, there is a difference in method. In Hearsay-II, several rival hypotheses may coexist at each level (a *hypothesis* being a possible interpretation of the data at a given level), and the program evaluates all of them. Copycat constructs only one view at a time (for example, at any given time, the first **c** in the string **abccc** can be seen either as the rightmost letter of the group **abc** or as the leftmost letter of the group **ccc**, but not both), but that view is malleable, and can be easily reshaped, given the right kinds of pressure. Humans cannot see the same high-level thing in two ways at once, but, as with the famous Necker cube, they can switch back and forth between coherent perceptions with varying degrees of ease. Thus Copycat's method is more psychologically realistic than that of Hearsay-II. The latter method suffers from a potential combinatorial problem: competing hypotheses can exist for different pieces of a whole, so the number of compound hypotheses at higher levels can become very large.

There is also a difference between the control structures of Copycat and Hearsay-II. In Hearsay-II, a central scheduler assigns a "priority" to each active knowledge source, the priority being an estimate of the likely usefulness of the knowledge source's action in fulfilling the overall goal of recognizing the utterance. The notion of priority is somewhat different from that of codelet urgency in Copycat. There is no randomness in Hearsay-II; the scheduler always chooses the highest-priority knowledge source to run. This reflects a difference in philosophy between the two programs. At each point, Hearsay-II tries to make an intelligent decision about what to do next: it uses global knowledge about the current state of the interpretation to assign priorities, and chooses what seems to be the overall best thing to do next. The control structure of Copycat is simpler; since an individual codelet cannot see globally and cannot make very intelligent decisions, the urgencies it assigns are based on local information, and since individual codelets do very small jobs, no single decision is very important. What to do next is decided probabilistically, and Copycat's overall "intelligence" emerges from the statistics of this probabilistic control structure. As was described in Chapter 3, Copycat's strategy of parallel and fine-grained exploration ensures *fairness* in deciding what should be explored, whereas always deterministically

choosing the *most promising* pathway (even when very little information has been obtained and one thus has very little confidence in one's assessment of promise) does not allows alternative views the chance to be developed. This could be seen clearly in the results of Experiment 6 in Chapter 7, in which the temperature was clamped at a very low value, and Copycat's control strategy became similar to that of an agenda system. The program became very conservative in what possibilities it explored, and it was unable to build the structures necessary to discover the interesting answer **mrrjjjj**.

8.2.4 Semantic Networks

A large number of artificial-intelligence computer programs have used semantic networks in order to represent knowledge. Copycat's Slipnet shares some features with standard semantic networks, but is, in many important ways, a quite different kind of structure. To point out some of the similarities and differences, I will compare the Slipnet with two examples of programs using semantic networks: Quillian's semantic-memory model (with extensions by Collins and Loftus), and Anderson's ACT*.

Quillian's semantic-memory program (Quillian, 1968) was the first formulation of what has now become commonly known as a semantic network. The network consisted of nodes representing English words and links representing relations between nodes. Quillian's focus was on modeling language understanding, and his network was designed to encode the meanings of words. Thus each node in the network corresponds to a word, and there are five different kinds of links: 1) superordinate links (e.g., "apple ISA fruit") and subordinate links (e.g., "fruit HAS-INSTANCE apple"); 2) links connecting nouns with modifiers (e.g.,"apple IS red"); 3) disjunctive sets of links (e.g., linking three different meanings of the word "plant"); 4) conjunctive sets of links (e.g., linking several necessary attributes of plants, such as "needs air", "needs water" and so on); and 5) links between subjects and objects (e.g., "person eats food"). Each link has an optional strength value associated with it, indicating how important the given relationship is to the meaning of the word.

A typical task in which this network was used is that of "comparing and contrasting" the meanings of a given pair of words. This is done by tracing out all paths in the network from each of the words and finding a point where two paths intersect. Because of the structure of the network, the two paths (up to the point of intersection) can be put in the form of English sentences. For example, when asked to compare the words "Earth" and "live", the program found an intersection at the node "animal", and the two paths leading up to that

node were expressed as "Earth is the planet of animal" and "To live is to have existence as animal". These two sentences constituted the program's response to the query.

The structure of Copycat's Slipnet is quite different from Quillan's semantic-memory network. Since Quillian's goal was to model language-understanding, the structure of his network roughly mimicked the structure of English, whereas the Slipnet is a model of associations and potential slippages. Thus the kinds of nodes and links included in each system are quite different. A concept in the Slipnet has blurry boundaries, defined by a probability distribution centered around a central node; in Quillian's network, concepts (such as *person* or *apple*) are atomic, and there is no notion of conceptual slippage. Patterns of activation in the Slipnet signify the relevance of the activated concepts, and the spread of activation is a model for how associations between related concepts come about. This is quite different from Quillian's notion of activation, which is used to trace out sentence-like pathways in the network, not to represent the degree of relevance of certain concepts. Finally, the Slipnet is meant to be a model of the adaptability of concepts, in that the activations of nodes and the distances between nodes change in response to what is being perceived. The links in Quillian's network can have strengths, but these are fixed ahead of time; the network does not respond to varying context. In fact, this is a major difference between the two systems; activities in the Slipnet are tightly interrelated with the perceptual activities of codelets, and the two sets of activities continually influence each other, whereas Quillian's network is used in isolation.

Collins and Loftus (1975) have proposed several extensions to Quillian's original network to allow it to model several experimental results on human memory. Four of these proposed extensions have features that are also included in Copycat's Slipnet. First, they propose that activation act like a signal from a source that attenuates (according to the strength of links) as it travels outward. This is similar to Copycat's spreading-activation mechanism. Second, Collins and Loftus propose that a node should continue to spread activation as long as it is being processed. This method is used in Copycat: a node stays activated (and thus spreads activation) as long as instances of it continue to be perceived. Third, Collins and Loftus suggest that, as in Copycat, the activation of a node should decay over time. Finally, they propose that the network be organized along the lines of semantic similarity, where conceptual relatedness between nodes is measured by the number of properties two nodes have in common. This idea gets closer to the associative structure of Copycat's Slipnet, although in the Slipnet, similarities between linked nodes are not always spelled out.

A more recent model involving a semantic network and spreading activation is John Anderson's ACT* (Anderson, 1983). ACT* is intended to be a theory of the "architecture of cognition": that is, of the basic principles of operation of cognitive systems. As one would expect with so ambitious a goal, Anderson's system is quite complex. It warrants more discussion than I have space for here; I will only outline the similarities and differences between the architectures of ACT* and Copycat.

ACT* has three memories: a *declarative memory* storing long-term declarative knowledge, a *working memory* containing the currently active parts of declarative memory, and a *production memory* containing the procedural knowledge of the system. The declarative memory is a semantic network in which each node is a cognitive unit: a sort of "unbreakable" unit of knowledge. A cognitive unit can be either a proposition (e.g., "Bill hates Fred"), an ordered string (e.g., "one, two, three"), or a spatial image (e.g., a triangle above a square). In the examples given by Anderson, the declarative memory consists of many cognitive units linked together encoding knowledge about specific situations. Cognitive units are thus very different from the nodes in Copycat's Slipnet—they are small pieces of specific knowledge rather than concepts—and ACT*'s declarative memory has nothing like the distributed concepts and context-dependent conceptual proximities in the Slipnet. However, there are some similarities between the two systems. As in Copycat's Slipnet, the nodes in ACT*'s declarative memory become activated by input to the system, spread activation to neighboring nodes, and lose activation through decay. As in Copycat, the activation of a node indicates its salience or relevance, and spreading activation is a parallel process that spreads relevance through the network.

All processing in ACT* is carried out by productions, which are activated by pattern-matching tests on the contents of working memory. The amount of time each test takes is determined by the strength of the production (a function of its prior success) and the activations of the declarative-memory nodes mentioned by the production. This accelerates the execution of certain types of productions in response to interacting pressures, giving ACT*'s production-matching mechanism something of the flavor of Copycat's parallel terraced scan, in which the activations of Slipnet nodes affect the urgencies of codelets, thereby translating pressures into differential rates of exploration of pathways.

ACT* has been used mainly in modeling high-level cognitive skills such as decision-making, mathematical problem-solving, computer programming, and language generation. Thus its focus is quite different from that of Copycat. This difference in focus is reflected

not only in the difference between ACT*'s declarative memory and Copycat's Slipnet, but also in the large difference between the kinds of actions performed by ACT*'s productions and those performed by Copycat's codelets. ACT*'s processing is directed by explicit, conscious goals, where a goal is represented by an active cognitive unit that dominates processing until the goal is accomplished. In contrast, goals in Copycat are emergent rather than explicit; they do not explicitly dominate processing, but act as top-down pressures for perceptual biases. For example, the goal "Look for successor groups" is implicit when the successor-group node is activated; high activation results in higher urgencies and more success for codelets trying to form successor groups. In addition, the examples given by Anderson show that ACT*'s productions are quite specific and domain-dependent, and the tasks performed are at a fairly high cognitive level, unlike the unconscious or subcognitive level of a codelet's task. For example, the following is one of the productions used in a geometry-proof generation task (the production has been translated into English by me): "If the goal is to prove a certain statement, and that statement is about a certain relation, and a certain postulate is also about that relation, and the teacher suggests that postulate, then set as a subgoal to try that postulate, and mark that postulate as tried." This is very different in flavor from the small perceptual activities of Copycat's codelets.

8.2.5 Connectionist and Classifier-System Models, and Copycat's Place in the Symbolic/Subsymbolic Spectrum

The philosophy behind the Copycat project is similar in many ways to that of various connectionist or "parallel distributed processing" (PDP) models (Rumelhart & McClelland 1986) and to that of classifier systems (Holland, 1986; Holland et al., 1986).

Connectionist networks are pattern-recognition and learning systems. A connectionist network consists of a number of nodes connected by weighted links. In such a network (at least in the ideal PDP variety), a single node does not symbolize anything on its own; rather, concepts and individual instances of concepts are represented as activation patterns distributed over large numbers of nodes. Typically, the nodes in the network are divided up into two or more layers, consisting of an input layer, possibly some internal "hidden" layers, and an output layer. The recognition process consists of presenting an activation pattern (representing an instance of something the network is supposed to recognize or categorize) to the input layer, and allowing this activation to spread throughout the network over the links as a function of the weights of the links. The "answer" (e.g., the category the input

instance belongs to) winds up being displayed as an activation pattern on the output nodes. As the network is given more and more input patterns, it gradually improves its performance as a result of a learning algorithm that adjusts the weights on the various links (there are many different such learning algorithms used in connectionist systems; the most common one is known as *back-propagation*, Rumelhart, Hinton, & Williams, 1986).

Classifier systems are also learning systems. A classifier system is composed of a large number of simple agents called *classifiers*. The system has an input interface and an output interface. Into the input interface come "messages" about the current state of the environment and the system's relation to it. The job of the classifiers is to classify messages—that is, to decide what to do in response to them. As in connectionist networks, the principles of self-organization and emergence are central to classifier systems: the representations of concepts and instances of concepts are at any time distributed over a number of classifiers. There is no Central Director controlling the actions of the system; rather, all of the system's behavior arises from myriad cooperative and competitive interactions among the individual classifiers. Classifiers that produce beneficial messages for the system tend to get stronger (via a credit-assignment procedure known as the "bucket brigade" algorithm) and thus are more likely to win competitions with other classifiers (such competitions are probabilistically decided on the basis of strength). Another learning mechanism, known as the "genetic algorithm", effects a kind of natural selection among classifiers in which weak classifiers die out and in which strong classifiers thrive and, via reproduction (involving recombination with other strong classifiers), pass their "genes" on to offspring classifiers. The combination of the credit-assignment mechanism and the genetic algorithm should in principle allow the system to adapt (via reapportionment of strength, deletion of unhelpful classifiers, and creation of new classifiers) to the environment it faces.

Connectionist networks and classifier systems are examples of *subsymbolic* (also called *subcognitive*) architectures. Smolensky (1988) characterizes the difference between the symbolic and subsymbolic paradigms as follows. In the symbolic paradigm, descriptions used in representations of situations are built of entities that are *symbols* both in the semantic sense (they refer to categories or external objects) and in the syntactic sense (they are operated on by "symbol manipulation"). In the subsymbolic paradigm, such descriptions are built of *subsymbols*: fine-grained entities (such as nodes and weights in connectionist networks or classifiers in a classifier system) that give rise to symbols. In a symbolic system, the symbols used as descriptions are explicitly defined (e.g., a single node in a semantic network

represents the concept "dog"). In a subsymbolic system, symbols are statistically emergent entities, represented by complex patterns of activation over large numbers of subsymbols. Smolensky makes the point that subsymbolic systems are not merely "implementations, for a certain kind of parallel hardware, of symbolic programs that provide exact and complete accounts of behavior at the conceptual level" (p. 7). Symbolic descriptions are too rigid or "hard", and a system can be sufficiently flexible to model human cognition only if it is based on the more flexible and "soft" descriptions that emerge from a subsymbolic system.

The faith of the subsymbolic paradigm is that human cognitive phenomena are emergent statistical effects of a large number of small, local, and distributed subcognitive events with no global executive. This is the philosophy underlying connectionist networks, classifier systems, and Copycat as well. Fine-grained parallelism, local actions, competition, spreading activation, and distributed and emergent concepts are essential to the flexibility of all three architectures (although in classifier systems, spreading activation is not explicit, but rather emerges from the joint activity of many classifiers). Some connectionist networks (e.g., Boltzmann machines, Hinton & Sejnowski, 1986, and Harmony-Theory networks, Smolensky, 1986) have an explicit notion of computational temperature with some similarity to Copycat's (though, as was explained in Chapter 3, there is a significant difference between the use of temperature in Copycat and in simulated annealing, which is essentially the temperature notion used by Hinton & Sejnowski and by Smolensky). In classifier systems, something akin to a parallel terraced scan emerges from probabilistically decided competitions among classifiers and from the genetic algorithm's implicit search through schemata (i.e., templates for classifiers) at a rate determined by each schema's estimated promise (see Holland, 1988, for a description of the dynamics of such searches in genetic algorithms). In addition, the interaction of top-down and bottom-up forces is central both in connectionist systems (see for example, McClelland and Rumelhart's model of letter perception, 1981) and in classifier systems (for example, as discussed in Chapter 2 of Holland et al., 1986).

The philosophy underlying the Copycat project is more akin to that of the subsymbolic paradigm than that of the symbolic paradigm, but the actual program fits somewhere in between. Concepts in subsymbolic systems are often highly distributed, being made up of individual nodes that have no semantic value in and of themselves, whereas in symbolic systems, concepts are represented as simple unitary objects (e.g., as Lisp atoms). Concepts in Copycat could be thought of as "semi-distributed", since a concept in the Slipnet is probabilistically distributed over only a small number of nodes—a central node (e.g., *suc-*

cessor) and its probabilistic halo of potential slippages (e.g., *predecessor*). The basic units of subsymbolic systems such as connectionist networks are meant to model mental phenomena further removed from the cognitive, conscious level than those modeled by Copycat's Slipnet nodes and codelets. It may be that these systems are more psychologically realistic than Copycat, but their distance from the cognitive level makes the problem of controlling their high-level behavior quite difficult, and I don't think that at this point it would be possible to use such systems to model the types of high-level behavior exhibited by Copycat. Ideally, a model should be constructed in which a structure such as Copycat's Slipnet arises from such a low-level, distributed representation, but this is beyond the achievements of current research in connectionism. Likewise, in classifier systems, several properties implanted directly in Copycat (such as nodes, links, and spreading activation) would have to emerge automatically, which I believe would make a high-level task, such as Copycat's, quite difficult for classifier systems as they are currently conceived. Thus Copycat models concepts and perception at an intermediate level, in terms of the degree to which concepts are distributed and the extent to which high-level behavior emerges from lower-level processes.

A major difference between Copycat's architecture and that of connectionist networks is the presence in Copycat of both a Slipnet, containing platonic concept *types*, and a working area, in which structures representing concept *tokens* (i.e., instances of concepts) are dynamically constructed and destroyed. Connectionist networks have no such separate working area; both types and tokens are represented in the same network. This has led to a great deal of research in connectionism on the so-called "variable-binding" problem, which is related to the larger question of the relationship between concept types and concept tokens. One reason researchers in connectionism may hesitate to make such a separation is that neural plausibility is a very important part of their research program, and a structure like Copycat's Workspace—a mental region in which representations of situations are constructed—does not have a clear neural underpinning. In contrast, for the purposes of Copycat and related projects, we are influenced more by psychological than neurological findings. We assume the existence of something like Copycat's Workspace even though we do not know its neural basis, and we investigate how a spreading-activation network with distributed concept *types* interacts with a working area in which ephemeral concept *tokens* can be arranged in complex structures. The lack of such a working area in connectionist networks is another reason why it may turn out to be very difficult to use such systems to model concepts and high-level perception in the way Copycat does.

In classifier systems, the "message list" (on which all messages from the environment and from classifiers are posted) roughly corresponds to Copycat's Workspace; messages can serve as the tokens corresponding to concept types (classifiers). It is possible that structures similar to those built in Copycat could be represented in a classifier system as messages on a message list, though precisely how to do this is an open question.

Connectionist networks and classifier systems learn from run to run, while Copycat does not. As was said before, Copycat is not meant to be a model of learning in this strict sense, though it does model some fundamental aspects of learning: how concepts adapt to new situations that are encountered, and how the shared essence of two situations is recognized.

The belief underlying the methodology of the Copycat project is that building a model at the level of Copycat's architecture is essential not only for the purpose of providing an account of the mental phenomena under study at its intermediate level of description, but also as a step towards understanding how these phenomena can emerge from even lower levels. The "subsymbolic dream"—that all of cognition can be modeled using architectures at the subsymbolic level of connectionist networks—may be too ambitious at this point in the development of cognitive science. If there is any hope for understanding how intelligence emerges from billions of neurons, or even how it might emerge from connectionist networks, we need to understand the intermediate-level mechanisms underlying the structure of *concepts*, a term referring to mental phenomena of central importance in psychology that nonetheless still lack a firm scientific basis. The long-term goal of the Copycat project and related research is to use computer models to help provide such a scientific basis. The hope is that the understanding that results from this approach will not only in its own right contribute to answering long-standing questions about the mechanisms of intelligence, but will also provide a guide to connectionists studying how such intermediate-level structures can emerge from neurons or cell-assemblies in the brain.

In summary, the architecture of Copycat is very different from the more traditional, so-called "symbolic" artificial-intelligence systems, both in its parallel and stochastic processing mechanisms, and in its representation of concepts as distributed and probabilistic entities in a network. These features make it more similar in spirit to connectionist systems, though again there are important differences. The high-level behavior of connectionist systems emerges statistically from a lower-level substrate as in Copycat. However, the fundamental processing units in connectionist systems are more primitive, concepts in such networks are distributed to a much higher degree than in Copycat, and concept types and tokens are

required to reside in the same network. Consequently, there has not been much success so far in using connectionist systems as models of high-level cognitive abilities such as analogy-making. Copycat thus explores a middle ground in cognitive modeling between the high-level symbolic systems and the low-level connectionist systems; the claim made by this research is that this level is at present the most useful for understanding the fluid nature of concepts and perception evident in analogy-making.

CHAPTER IX

CONCLUSION

In this chapter, I first summarize the main points of this dissertation, and then present some proposals for future work on this project. Finally, I discuss the contributions of this project to research in cognitive science and artificial intelligence.

9.1 Summary of Dissertation

This dissertation has presented the work done so far on the Copycat project, an investigation of high-level perception, conceptual slippage, and analogy-making in humans. The long-term goal of this project is to understand the mental mechanisms underlying the flexibility and adaptability of concepts and of perception, particularly as they are manifested in the context of analogy-making. The point of this section is to highlight and summarize the major ideas presented in this dissertation in order to give the reader a clearer perspective on what has been accomplished. The summary will be given chapter by chapter.

In Chapter 1, the terms "high-level perception" and "conceptual slippage" were defined, and the relationship among various aspects of high-level perception—categorization, recognition, and analogy-making—was discussed. Many examples were given to illustrate the blurry boundaries between these various mental activities and to support the claim that these activities arise from similar mental mechanisms. In particular, central to all of them is the phenomenon of conceptual slippage, in which some mental representations are not held fixed but are allowed to be replaced by conceptually related descriptions in response to pressure. The central feature of high-level perception is the fluid application of one's existing concepts to the different situations one encounters, and conceptual slippage is required for this fluidity. The examples given in Chapter 1 demonstrated how conceptual slippage

shows up particularly clearly in the realm of analogy-making, and illustrated the necessity and ubiquity of conceptual slippage and analogy-making at all levels of thought.

In Chapter 2, Copycat's letter-string microworld was presented, and examples were given to support the claim that this microworld captures, in an idealized form, many essential issues in high-level perception and analogy-making. A number of general abilities necessary for analogy-making (both in the microworld and in the real world) were discussed. Chapter 2 also proposed two types of criteria for judging Copycat's success, reflecting the program's interdisciplinary goals: artificial-intelligence criteria, which focus on the range of problems the program can deal with, and psychological criteria, which focus on more detailed comparisons of Copycat's behavior with that of people.

In Chapter 3, Copycat's architecture was described, and the proposed mechanisms for achieving the abilities listed in Chapter 2 were detailed. The major parts of Copycat's architecture are:

- *The Slipnet*, in which a concept consists of a central region (represented by a node) surrounded by a halo of potential associations and slippages (represented by neighboring nodes linked to the central node). Since activations and link-lengths vary dynamically, and since concepts are probabilistically rather than explicitly defined, the availability and relevance of concepts in the Slipnet and their degree of association with other concepts change as perception and analogy-making proceed, and the network gradually settles into a state that reflects essential properties of the situation at hand. In other words, the Slipnet as a whole fluidly adapts to the different situations the program is presented with.

- *The Workspace*, which is meant to correspond to the mental area in which ephemeral representations of situations are constructed and destroyed.

- *Codelets*, which scout out, evaluate, and build (and sometimes destroy) structures representing the program's interpretation of the problem at hand. Different codelets correspond to different types of structures as well as to different *pressures* in an analogy (e.g., a pressure to find in one situation the counterparts of *important* entities in the other situation, or a pressure to perceive instances of a particular concept, such as *successor group*). There are bottom-up codelets, which represent pressures present in *any* situation, and top-down codelets, which represent pressures specific to the situation at hand.

- *Temperature*, which measures the degree of perceptual disorganization in the system, and in turn controls the degree of randomness used in making decisions.

Fundamental to Copycat is the notion of *statistical emergence*: the program's high-level behavior emerges from the interaction of large numbers of lower-level activities in which probabilistic decisions are made. Codelets, nodes, and links are all defined explicitly ahead of time, but their interaction gives rise to three types of statistically emergent entities: (1) emergent *concepts*, whose composition (in terms of the nodes that are included) and whose availability and relevance to the situation at hand are statistical (rather than explicitly defined) properties; (2) emergent *pressures*, which arise as statistical effects of large numbers of codelet actions; and (3) an emergent *parallel terraced scan*, which results statistically from a large number of probabilistic choices based on codelet urgencies and other factors (e.g., salience of objects, strengths of structures, etc.).

These three types of emergent entities interact as well. The structure and activation of concepts influences both how codelets will evaluate possible structures (a codelet's evaluation of a structure almost always takes into account activations and conceptual distances in the Slipnet) and which top-down codelets will be posted. Concepts thus affect the population of codelets in the Coderack and their urgencies, out of which arise statistical pressures and a parallel terraced scan. The parallel terraced scan, by guiding the search through possible structurings of the problem, affects the activations of nodes and thus the conceptual distances encoded by links in the Slipnet. This interaction has the flavor of Hofstadter's vision of "emergent symbols" in the brain, in which the top level (the symbolic level) reaches back down towards the lower levels (the subsymbolic levels) and influences them, while at the same time being itself determined by the lower levels (Hofstadter, 1979, Chapter 11). This kind of system, in which explicitly defined entities (e.g., codelets, nodes, and links) give rise to implicit higher-level patterns (e.g., concepts, pressures, and the parallel terraced scan), which in turn reach back and influence the lower levels and thus each other, is an example of Forrest's (1990) characterization of "emergent computation" .

This interaction gives rise to a system in which concepts and perceptual exploration fluidly adapt to the situation at hand, and allow appropriate conceptual slippages to be made. Temperature-controlled nondeterminism is an essential component of Copycat. It allows the system to gradually shift from being parallel, random, and dominated by bottom-up forces to being more deterministic, serial, and dominated by top-down forces as the system gradually closes in on an appropriate way of conceiving the situation, which yields a

solution to the problem posed. The fact that the composition and activation of concepts, the type and strength of various pressures, and the parallel terraced scan emerge statistically from large numbers of probabilistic decisions imbues Copycat with both flexibility and robustness. Because of nondeterminism, no path of exploration is absolutely excluded *a priori*, but at the same time, the system has mechanisms that allow it to avoid following bad pathways, at least most of the time. A crucial idea is that the program has to have the *potential* to follow risky (and perhaps farfetched or even crazy) pathways in order for it to have the flexibility to follow subtle and insightful ones. This was strikingly illustrated by the results of Experiment 6 in Chapter 7, in which nondeterminism was basically eliminated. The program no longer gave farfetched fringe answers to "**abc** ⇒ **abd**, **mrrjjj** ⇒ ?", but it also no longer gave the insightful answer **mrrjjjj**. The program has to have the potential to bring in *a priori* unlikely concepts (such as *group-length*) into its interpretation of the problem, but should do so only in response to strong pressures. These pressures are what give shape to the program's concepts and guide the program's exploration.

Chapters 4 and 5 presented the major empirical results of the Copycat project.

In Chapter 4, statistics were given for Copycat's performance on the five target problems discussed in Chapter 2, and, for each problem, comparisons were made between the program's range of answers and the range of answers given by people participating in a survey. In addition, sets of screen dumps from runs of the program on each problem were given, which illustrated the mechanisms described in Chapter 3.

Chapter 5 addressed one of the most important questions for any artificial-intelligence program: How flexible is it? That is, how well does it continue to perform when it is stretched beyond the most central problems it was deliberately designed to solve? In this chapter, statistics summarizing the program's performance on 27 variants of the five target problems were given. The program's answers were again compared with the results of a survey of people's answers.

Chapter 6 gave a discussion of two salient ways in which the program is lacking: its mechanisms for implementing top-down forces and focus of attention are not effective enough, and it lacks sufficient self-watching mechanisms.

Chapter 7 gave the results of six experiments on the program designed to elucidate the roles played by various aspects of the program's architecture. In each experiment, a certain major design feature of the program was "lesioned" (i.e., removed or altered). The experiments investigated the effects on the program's behavior of suppressing or altering the

following features: terraced scanning, breaker codelets, different conceptual-depth values, dynamic link-lengths, and randomness (by clamping the temperature at either a very high or very low value).

Finally, in Chapter 8, Copycat was compared with some other models of analogy-making and with other related work in artificial intelligence and cognitive science, and a discussion was given of Copycat's intermediate place in the subsymbolic-to-symbolic spectrum of cognitive models.

9.2 Proposals for Future Work

As I see it, there are three dimensions along which future work related to this project could proceed. Work could continue on the Copycat program itself, addressing some of the problems with the current program that were discussed earlier, and the program could also be extended to deal with a larger set of problems in the letter-string domain as well as to produce more complete sets of answers to the problems it can currently solve. Another dimension of future work is to use the same basic architecture in other microdomains of roughly the same complexity. A third dimension would be to use ideas from Copycat to develop AI and cognitive-science models that work in more complex domains.

As far as future work on the Copycat program goes, perhaps the first priority is to address the problems with top-down forces, focus of attention, and self-watching that were discussed in Chapter 6. Having more plausible and effective mechanisms in these areas is essential in order to further extend the program.

Extensions to the program could be made in many directions. Every type of structure the program builds could be made more complex. The following are some examples of the kinds of extensions that could be made.

- More complex descriptions could be made, such as "rightmost letter of leftmost group" (e.g., the rightmost **a** in **aaabbbccc**), "third letter from the leftmost letter of string" (e.g., the **r** in **pqrst**), "next-to-leftmost letter" (e.g., the **j** in **ijklm**), and "next-to-last letter (in the alphabet)" (a possible description of the **y** in **wxy**).

- More complex bonds could be constructed, such as allowing bonds simultaneously between letter-categories and group-lengths in **rssttt**, bonds between spatially non-adjacent letters such as the **a** and **b** in **axbxcx**, and bonds representing relationships consisting of a *chain* of links rather than just one link in the Slipnet, such as the

"double successor" relations in **ace**.

- More complex types of groups could be constructed, such as groups based on spatial adjacency rather than on relations in the Slipnet (e.g., the three **mxb** groups in the string **mxbmxbmxb**), or groups based on symmetry (such as a grouping of the whole string **axxgggxxa**).

- More complex correspondences could be constructed, such as the three different correspondences from the **c** in the initial string to the rightmost letter in each successor group of the target string in "**abc** ⇒ **abd**, **lmnfghopq** ⇒ ?" (Variant 13). Another extension related to correspondences would be a mechanism that carries out the "coattails effect" mentioned in Chapter 7. The coattails effect should allow certain slippages to be pulled along "on the coattails" of conceptually related slippages that have already been made. This would enable the program to produce additional answers to problems such as Variant 25: "**abc** ⇒ **abd**, **glz** ⇒ ?". The current program cannot answer **flz**—there is no possibility of a *successor* ⇒ *predecessor* slippage, since the target string cannot be seen as a successor or predecessor group. The coattails effect would allow the slippage *successor* ⇒ *predecessor* to be brought in on the coattails of the conceptually related slippage *first* ⇒ *last*, even though the former is not explicitly a part of any correspondence.

- More complex rules could be constructed, such as "Extend the string by one" for initial change **abc** ⇒ **abcd**, or "Replace all letters by *X*'s" for initial change **abc** ⇒ **xxx**, or "Extract the rightmost letter" for initial change **abc** ⇒ **c**.

These are some of the ways in which the current program could be extended. This list is by no means exhaustive; in fact, it barely scratches the surface of what could be done. The letter-string domain has the potential for so many different types of problems that there is almost no limit to the kinds of extensions that could be made to Copycat. Much could be learned about the general issues we are studying by attempting to extend Copycat in various ways. (See Appendix A for a number of examples of problems Copycat cannot currently solve, some of which suggest other possible extensions.)

The second dimension for future work is to use the same basic architecture in other microworlds. Attempting to use the same architecture in different contexts would be very useful for learning what aspects of the architecture are (perhaps inadvertently) domain-dependent and for determining how to make the architecture more general. Such a project

is currently being carried out by Robert French. In his "Tabletop" project, he uses a similar architecture to solve analogy problems involving arrays of objects on a restaurant table. There are some interesting differences between the issues contained in the Tabletop domain and those in the letter-string domain, and this work will hopefully result in a more refined and general architecture. (For a description of the Tabletop project, see Hofstadter, Mitchell, & French, 1987.)

Another, more ambitious project (which has not yet reached the implementation stage) is Hofstadter's "Letter Spirit" project (also described in Hofstadter, Mitchell, & French, 1987), which proposes to use an architecture related to Copycat's to produce "gridfonts" (typefaces in which all letters are designed on a specific grid, using discrete straight segments rather than continuously curving lines) in uniform styles. For example, the input to the program might be an 'a' drawn on the grid by a person; the program's task would be to produce the rest of the alphabet in "the same style". As was discussed in Chapter 1, the process of recognizing or producing certain styles (e.g., in music, art, or typography) is basically a process of analogy-making. Here the analogy problem is, given an 'a', to produce an analogous 'b', 'c', and so on.

The third dimension for future work is to attempt to use general ideas from Copycat (such as the notions of the parallel terraced scan, context-dependent concept activations and conceptual distances, probabilistically defined concepts with graded presence or relevance, temperature, etc.) in computer models of perception working in more complex domains. For example, I think it would be of great interest to try to use these ideas in computer models of real-world visual and auditory recognition processes, such as object recognition or speech understanding. I am not going to make any specific proposals here for how this might be accomplished, but I feel that the ideas in Copycat are now well-enough developed so that such applications could begin to be considered. All of the main mechanisms in Copycat were designed and implemented as much as possible with the idea that they could be eventually "scaled up"—in principle, they do not rely on the fact that there are only a small number of elements in each problem or a small number of nodes and links in the Slipnet. In practice, there will no doubt be difficulties in actually getting these mechanisms to work in more complex situations. However, as I will discuss further in the next section, it is essential to attempt to do so, because confronting these very difficulties is what will lead to new insights about the issues Copycat is meant to address.

9.3 Contributions of This Research

> Until words like "concept" have become terms as scientifically legitimate as, say, "neuron" or "cerebellum", we will not have come anywhere close to understanding the brain. (Hofstadter, 1985c, p. 234)

> No one knows how to represent a concept or thing on the subsymbolic microlevel, or even precisely what this means. If this ambition could be recognized, it would come close to cracking the cognition problem. And no one is close to accomplishing that. (Pagels, 1989, p. 141)

These sentiments reflect the view that underlies the research described in this dissertation—that the understanding and explication of the psychological notion of "concept" is perhaps the most important problem facing cognitive science. Concepts can be said to be the fundamental units of thought, as genes are the fundamental units of heredity. And the present state of cognitive science is something like the state of biology before the recognition of DNA as the hereditary substance: the notion of a "gene" existed, but it was a vague and proto-scientific term awaiting an explanation in terms of lower-level biological entities and mechanisms. Likewise, the brain mechanisms underlying concepts are not currently known. It seems likely that a full account of these mechanisms will be much more complex and much more difficult to uncover than was the account of genes in terms of DNA.

As was said in the previous chapter, the long-term goal of the Copycat project (and related projects) is to use computer models to help provide such a scientific basis for concepts. This goal is still quite distant. An early step in this process would be to make clearer what concepts are in a psychological sense (as opposed to a neurological sense) and to elucidate the issues surrounding them. As was mentioned in Chapter 1, there has been much research in psychology on the *internal structure* of categories, and much light has been shed on those issues. The focus of the Copycat project is somewhat different: we are concentrating on investigating and clarifying the nature of conceptual slippage and the dynamics of the activation and association of concepts as they interact with perception. The hope is that the research described here has contributed to the understanding of these aspects of concepts.

As part of this process of elucidation, this dissertation has described a set of ideas about concepts, perception, and analogy-making, and has shown that a computer program that implements these ideas exhibits rudimentary fluid concepts—the program's concepts are able to adapt to different situations in a microworld that, though idealized, captures much of the essence of real-world analogy-making. The main contributions of this research have been to develop and explicate these ideas, to show the extent to which they do indeed

work, and also to examine in what ways they are flawed or incomplete. The result is not yet a complete "theory" of high-level perception and conceptual slippage, at least not in the standard sense of theories in, say, physics or chemistry. These ideas have not yet been sufficiently developed or implemented to be predictive of human behavior on a large scale, or to be strictly "falsifiable" (though some of the results given in this dissertation have demonstrated certain problems and incompletenesses in the program's mechanisms). Given that we are investigating very general abilities (rather than domain-specific performance on letter-string analogy problems), and that we are trying to understand how high-level mental activities (such as concepts) emerge from lower levels, the phenomena that we are studying are too complex for us to develop complete theories of them at this stage of research. Rather, the process of developing these ideas and implementing them in computer programs allows us to clarify what it is we are studying, and to begin to see what components such complete theories might have. The ideas presented in this dissertation are meant to act as stepping-stones for the development of more complex models and more complete theories.

This kind of approach is typical of the role of current computer models of intelligence. As Alfred Kobsa points out, "AI modeling certainly does provide us with deeper experience in recognizing what makes it possible for a system to produce certain 'intelligent' behavior. It can be assumed that efforts to make this background experience explicit and to state it in the form of generalizations will eventually lead to theories." (1987, p. 187).

For the purposes of this explication process, the importance of actually writing a computer program and getting it to work cannot be overemphasized. Many ideas for the Copycat program were originally set forth by Hofstadter in a broad, outline form (1984a) before the program was written, but it was the process of writing the program—requiring constant confrontation with the all-important "details"—that allowed the original ideas for the program to become more fully developed by Hofstadter and myself. Along the way, vague ideas were clarified, wrong ideas were discarded, and new ideas were added. This parallel development of ideas and models is how cognitive modeling has to proceed. Many insights can come only through grappling face to face with the real issues (and it is certain that this same process of modification will occur in attempts to use ideas from Copycat in modeling perception in more complex domains). New insights often come when things go wrong. In the process of writing the Copycat program, many unanticipated problems arose (such as the problems with top-down forces, focus of attention, and self-watching discussed in Chapter 6) that guided the implementation and that sometimes helped shed light on deep issues

in perception and analogy-making. The appearance of unexpected strange answers (such as "**abc ⇒ abd, hhwwqq ⇒ hhxxrr**", discussed in Chapter 5) demonstrated not only that something was wrong with the program, but also helped drive home the subtlety and complexity of the mental phenomena that we are studying. The process of writing a program such as Copycat brings up at least as many questions as it answers, and one of the points of writing the program was to find out what these questions *are*. There are many ways in which the current version of Copycat is lacking, and many problems with the mechanisms the program does have, and a result of writing the program is that these aspects and problems are uncovered and brought to light; they would have remained unseen and obscured if the program had never been written. (Some of these points concerning the advantages of writing artificial-intelligence programs have also been made by Longuet-Higgins, 1981, among others.)

In summary, the main contributions of this work are: clarifying and making explicit many central features of concepts, high-level perception, and analogy-making (e.g., emergent concepts, conceptual slippage, the interaction of bottom-up and top-down forces, commingling pressures, the parallel terraced scan, the role of nondeterminism in thought, etc.), presenting ideas for mental mechanisms underlying these features, and verifying and further developing these ideas by implementing them in a computer program. My hope is that the ideas and results described in this dissertation have fulfilled what I have asserted to be the main criteria for success—to help us to better understand what concepts are and to broaden our intuitions on how to think about these issues. All of this serves to set the stage for the very long-term goal of developing more complete scientific theories that explain how human cognition comes about in the brain and that propose how human-like intelligence might be achieved in computers.

APPENDICES

APPENDIX A

A Sampler of Copycat Analogies

A.1 Problems

The following is a collection of analogy problems in the Copycat letter-string domain (not including the five target problems and 27 variants given in Chapters 4 and 5). These are all problems that are currently beyond Copycat's abilities. The purpose of presenting them is to give readers a better feel for the breadth and richness of this microworld.

The problems given below are arranged into seven "families", each having a common idea or theme among its problems. All the problems are of the form "If S1 changes to S1′, how does S2 change?". In many of the families, several problems in a row are based on a single example. In those cases, the example is given only once, and the various targets are listed below it.

In the next section, the problems given here are discussed, and reasons are given for why Copycat cannot currently solve them.

1. **abc ⇒ abd**

a. **ace ⇒ ?**

b. **aababc ⇒ ?**

c. **pxqxrxsx ⇒ ?**

d. **aaabbbcck ⇒ ?**

e. **bcdacdabd ⇒ ?**

---

2. **abcd ⇒ abcde**

a. **ijklm ⇒ ?**

b. **ijxlm ⇒ ?**

c. **mlkji ⇒ ?**

d. **iiii ⇒ ?**

e. **iiiijjjj ⇒ ?**

---

3. *a.* **mmmkooeeeeefqxx ⇒ kfq**
riipppppplooyg ⇒ ?

b. **rrccmmkppbb ⇒ k**
ljooooosrezv ⇒ ?

---

4. *a.* **abcde ⇒ xxxxx**
pqr ⇒ ?

b. **xxh ⇒ fgh**
pxxx ⇒ ?

c. **pqrxxxx ⇒ pqrstuv**
efghmm ⇒ ?

d. **amcmemg ⇒ abcdefg**
wxyx ⇒ ?

---

5. *a.* **eeeeqee ⇒ eeeeree**
sosss ⇒ ?

b. **eeeqee ⇒ qqqeqq**
sabsss ⇒ ?

c. **eqe ⇒ qeq**
abcdcba ⇒ ?

d. **eqe ⇒ qeq**
aaabccc ⇒ ?

---

6. *a.* **abcdde** ⇒ **abcde**
pqstu ⇒ ?

b. **abced** ⇒ **abcde**
ppqqrrs ⇒ ?

---

7. *a.* **a** ⇒ **z**
b ⇒ ?

b. **pqr** ⇒ **rqp**
a ⇒ ?

A.2 Discussion

1*a.* "**abc** ⇒ **abd**, **ace** ⇒ ?". As was discussed in Chapter 2, the target string in this problem can be understood as a "double successor" group, yielding answer **acg**. Copycat currently can perceive relations or make slippages involving nodes separated by only one link in the Slipnet, so it cannot perceive a double-successor relation. In order to solve this problem, Copycat would have to perceive these relations and use them to create a new concept—*double successor*—on the fly. This new concept would have the same properties as the program's other concepts: it would be used by codelets to calculate strengths of structures involving it, when active it would post top-down codelets to look for instances of it, and so on. The program does not currently have any mechanism for creating new concepts such as this.

1*b.* "**abc** ⇒ **abd**, **aababc** ⇒ ?". As was discussed in Chapter 2, if the target string is parsed as **a-ab-abc**, then a strong though abstract similarity to the initial string **abc** emerges, where the "rightmost letter" of **aababc** is the group **abc**, and its "successor" is **abcd**, yielding answer **aababcd**. Copycat can solve this problem *in principle* in the same way it solves "**abc** ⇒ **abd**, **mrrjjj** ⇒ ?", but in practice it is too hard. The program usually very quickly constructs a sameness group consisting of the leftmost two **a**'s, and cannot break it in order to come up with the parsing **a-ab-abc**. The program usually answers **aababd** (using the rule "Replace rightmost letter by successor"), though it sometimes constructs an **abc** successor group from the rightmost three letters in the target string, and maps the **c** in the initial string onto this group, answering **aabbcd**. On occasion, the program

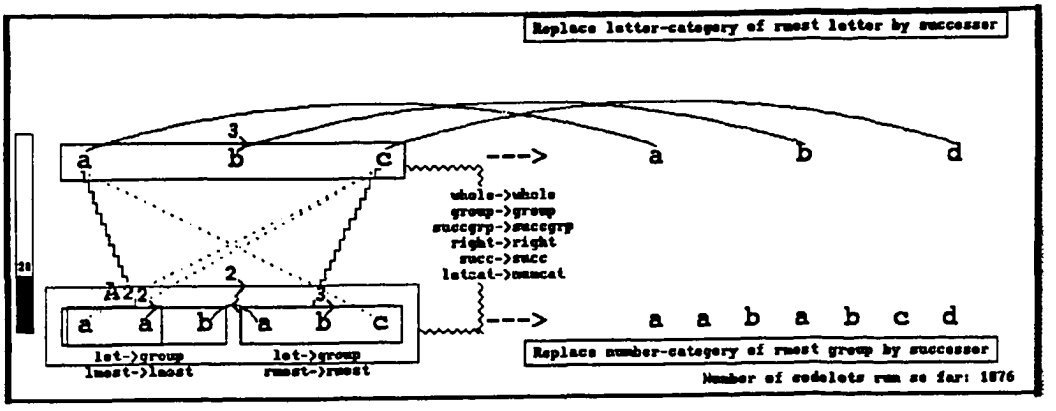


Figure A.1: Final configuration of the Workspace for a strange route to the answer **aababcd**.

gives the answer **aababcd** for the wrong reason: The **aa** group is constructed and then it is bonded together with the adjacent **b**, and a two-element successor group is formed: **aa-b**. The program also forms the target-string **abc** three-element group, and then notices a successor relation between the lengths of these two groups. It thus parses the string as **2-3** (i.e., **Ab-abc**) and answers **2-4**, that is, **aababcd**, as shown in Figure A.1.

This crazy "misspun tale" was the only way the program ever got the answer **aababcd** over thousands of runs. The reason in part has to do with the program's problems with top-down forces discussed in Chapter 6. Once the **abc** successor group is made in the target string, the combination of top-down forces and high temperature should ideally combine to make it more likely for a proposed **ab** group (i.e., containing the second and third letters) to successfully compete with the intrinsically strong **aa** group. Once the string has been parsed as **a-ab-abc**, then the same kinds of forces as are present in "**abc** ⇒ **abd**, **mrrjjj** ⇒ ?" should make it possible for the leftmost **a** to be seen as a single-letter *successor* group and for length relations to be noticed. This is all possible in principle for the current version of Copycat. What is not possible for the current program is to see *letter-category* relations between the groups **a**, **ab**, and **abc**, or concept-mappings based on letter-categories between the letters in the initial string and these three groups. These relations and concept-mappings would be based on the view that **a-ab-abc** was basically **A-B-C** in *code*, in the same way that **ii-jj-kk** is basically **I-J-K** in code. However, Copycat does not currently give letter-category descriptions to successor and predecessor groups as it does to sameness groups (e.g., the group **ii** is given the description "*letter-category*: **I**", but the group **abc**

is not given any letter-category description), so such bonds and concept-mappings cannot be made, although I believe they are in part what make the solution **aababcd** seem very strong to many people.

1*c*. "**abc ⇒ abd, pxqxrxsx ⇒ ?**". To solve this problem, once must separate the "figure" (the successor group **pqrs**) from the "ground" (the interleaved **X**'s). If this figure and ground are perceived, then the answer is **pxqxrxtx**. A more literal answer is **pxqxrxsy**. Copycat cannot solve this problem for several reasons. For instance, it cannot build bonds between letters that are not spatially adjacent (e.g., the **p** and **q** here are separated by an **x**) and it can only make groups that are based on relations in the Slipnet (thus it could not group the **p** and its neighbor **x** together as one unit).

1*d*. "**abc ⇒ abd, aaabbbcck ⇒ ?**". Here the question is what to do about that pesky **k**. Some possible answers are **aaabbbddd**, **aaabbbddk**, or **aaabbbccl**. The present version of Copycat can produce the last two (as well as the usual "Replace rightmost letter by **D**" answer, **aaabbbccd**), but it isn't flexible enough to do what people could do: assume that the **k** "really should have been a **c**", because then the analogy would make more sense, and given the answer **aaabbbddd**.

1*e*. "**abc ⇒ abd, bcdacdabd ⇒ ?**". This problem looks chaotic and senseless, unless you notice that the target string can be parsed as **bcd-acd-abd**, where each triplet is a code for the "missing" letter. When the string is thus decoded, it is simply **abc**, so the answer is **abd**, but once again in code: **bcd-acd-abc**. This is a very hard problem for people, and I don't know if Copycat will ever be sophisticated enough to get this answer.

.....

2. This family of problems shows various ways of *extending* a group. The current version of Copycat cannot deal with any of these problems, since it cannot yet form a rule like "extend the successor group".

2*a*. "**abcd ⇒ abcde, ijklm ⇒ ?**". The most straightforward answer is **ijklmn**.

2*b*. "**abcd ⇒ abcde, ijxlm ⇒ ?**". There are several possible answers, including **ijxlmn** (ignore the **x** and the lack of a **k**), **ijkxlmn** (viewing the target string as *two* successor groups separated by an **x**, both of which should be extended), and **ijklmn** (viewing the **x** as "the space into which to extend the group"). The last answer requires a kind of flexibility that is far beyond Copycat's current abilities.

2*c*. "**abcd ⇒ abcde, mlkji ⇒ ?**". The main rival answers here are **mlkjih** (extending

the *predecessor group* to its right) and **nmlkji** (extending the *successor group* to its left). These are similar to the rival answers to "**abc ⇒ abd, kji ⇒ ?**".

2*d*. "**abcd ⇒ abcde, iiii ⇒ ?**". If the concept of *successor group* is slipped to the concept of *sameness group*, or is generalized to the more general concept of *group*, then the answer is **iiiii** (that is, the group length is incremented by 1). Another reasonable answer is **iiijjjj**, perceiving the string **iiii** as a successor group of length 1 (consisting only of the chunk **iiii**), and extending it by one "letter" (here a group of four **j**'s). These answers (in my opinion) are much better than more literal-minded answers such as **iiij** (tack on the successor of the rightmost letter), **iiiie** (add on an **e** at the right), or **iiii** (do nothing, since the target string **iiii** contains no successor groups).

2*e*. "**abcd ⇒ abcde, iiijjjj ⇒ ?**". Two good answers are **iiiijjjjj** (extending both sameness groups at once) and
iiijjjjkkkk (extending the successor group seen at the group level).

.....

3. These problems involve the notion of *extracting* letters from a string, which Copycat does not currently have.

3*a*. "**mmmkooeeeeefqxx ⇒ kfq, riipppppplooyg ⇒ ?**". A plausible rule is "extract all *single* letters", yielding answer **rlyg**.

3*b*. "**rrccmmkppbb ⇒ k, ljooooosrezv ⇒ ?**". Here there is competition between two rules: the rule "extract all isolated letters" (yielding answer **ljsrezv**) or the even more abstract rule "extract the 'oddball' or 'black sheep'" (yielding answer **ooooo**). Giving Copycat the flexibility to recognize instances of the concept 'oddball' (or "objects in the situation that are different from all the other objects) in a psychologically plausible way would be extremely challenging. This is the kind of commonsense notion that people can use very easily and flexibly, but that would be very hard to impart to a computer program.

.....

4*a*. "**abcde ⇒ xxxxx, pqr ⇒ ?**". A plausible rule is "Replace all letters by *X*'s". This would yield the answer **xxx**. Copycat cannot currently construct rules describing changes of more than one letter, so it could not solve this problem.

4*b*. "**xxh ⇒ fgh, pxxx ⇒ ?**". This problem is easy for people, who answer **pqrs**. However, Copycat is quite far away from being able to perceive the relation between **xxh** and **fgh**, and to deal with the abstract concept of "filling in the spaces".

4*c*. "**pqrxxxx ⇒ pqrstuv, efghmm ⇒ ?**". A reasonable way of solving this problem is to see the **m**'s in **efghmm** as playing the same role as the **x**'s in **pqrxxxx**. This view would yield the answer **efghij**.

4*d*. "**amcmemg ⇒ abcdefg, wxyx ⇒ ?**". This problem has an amusing twist: the **x**'s in **wxyx** play the same "mask" or "placeholder" roles as the **m**'s in **amcmemg**, but what replaces the leftmost **x** in the desired answer (**wxyz**) is another copy of **x**, this time playing the role of itself! This problem, like the previous two, is far beyond Copycat's current capabilities.

.....

5*a*. "**eeeeqee ⇒ eeeeree, sosss ⇒ ?**". A reasonable rule here is "Replace the isolated letter by its successor", yielding answer **spsss**. Copycat cannot currently describe something as "the isolated letter", so it cannot at present form this rule.

5*b*. "**eeeqee ⇒ qqqeqq, sabsss ⇒ ?**". One very abstract rule is "switch the letters" (or "flip the bits"). If the **a** and **b** in the target string are grouped as a single unit, then the answer is **absababab**. Another way to describe the change is "turn the string inside-out". In this case, **assbbb** is a plausible answer. This problem is, of course, far beyond Copycat's current capabilities, since it has no notion of switching letters or turning things inside-out.

5*c* and *d*. "**eqe ⇒ qeq, abcdcba ⇒ ?**" and "**eqe ⇒ qeq, aaabccc ⇒ ?**". Both these problems explore the concept of "turning a string inside-out". A possible answer to part *c* is **dcbabcd**; possible answers for part *d* are **baaacccb, bbbacbbb,** and **abbbc**. The last answer expresses an extremely abstract view, in which the letters themselves are ignored and "turning the string inside-out" is done at the level of group-length. That is, the pattern 3-1-3 (**aaabccc**) corresponds to **eqe** and the pattern 1-3-1 (**abbbc**) corresponds to **qeq**.

.....

6. *a* and *b*. "**abcdde ⇒ abcde, pqstu ⇒ ?**" and "**abced ⇒ abcde, ppqqrrs ⇒ ?**". Both problems could be seen to be about "fixing up" or "cleaning up" a structure—a quite abstract concept (it would be very challenging to enable Copycat to recognize instances of it). A good answer to part *a* is **pqrstu**, and a good answer to part *b* is **ppqqrrss**. This is, in my opinion, somewhat better than the answer **pqrs**, since it reflects the idea that both the initial and target strings had "just a little bit wrong with them", rather than a massive defect requiring global rewriting.

.....

7a. "**a** ⇒ **z**, **b** ⇒ ?". If the **a** and the **z** are seen as "mirror images" of each other, then the answer should be **y**—that is, the **b**'s mirror image. For this answer, the rule would be something like "Replace the first letter by the last letter", and the translated rule would be something like "Replace the next-to-first letter by the next-to-last letter". Copycat cannot get this answer because it cannot presently describe a **b** as "next-to-first" or a **y** as "next-to-last".

7b. "**pqr** ⇒ **rqp**, **a** ⇒ ?". One answer is **a**. A very abstract answer is **z**, which might come about by seeing the essential relation between **pqr** and **rqp** as *opposite*, and asking, "What is *A*'s opposite?" This would involve something like the "coattails" effect suggested in Chapter 7. The **pqr** ⇒ **rqp** correspondences might involve the slippage *right* ⇒ *left* or alternatively *successor* ⇒ *predecessor*. Then the slippage *first* ⇒ *last* might come on the coattails of the other slippage, since the two slippages are conceptually related. Copycat currently has no mechanism implementing such an effect.

APPENDIX B

Parameters and Formulas

This appendix lists the values of the parameters used in Copycat, and gives more detailed descriptions of some of the formulas used in the program. The detailed formulas for the importance, happiness, and salience of objects, and for the strengths of structures, are not given here (they were described in general terms in Chapter 3), but the original source code can be provided to anyone interested in the details of these particular formulas.

All values in Copycat (parameters values, formula results, activation values, temperature, and so on) are in the range from 0 to 100.

There are many parameters in the program whose values were assigned by me. In general, the values were decided by a combination of intuition, trial and error, and some arbitrariness, and are not necessarily optimally tuned in the current version of the program. They should thus not be thought of as cast in concrete, but are very much open to further testing and refinement.

B.1 Values Used in Setting Up the Slipnet

B.1.1 Conceptual-Depth Values

A, *B*, ..., *Z*: 10

1, *2*, ... *5*: 30

leftmost, *rightmost*, *middle*, *single*, *whole*: 40

left, *right*: 40

predecessor, *successor*, *predecessor-group*, *successor-group*: 50

sameness, *sameness-group*: 80

first, *last*: 60

identity, *opposite*: 90

letter: 20

group: 80

letter-category: 30

number-category: 60

string-position: 70

direction: 70

bond-category: 80

group-category: 80

alphabetic-position: 80

object-category: 90

1.1.2 Link-Lengths

The length of a link in the Slipnet is determined by its label, if it has one, and otherwise is set ahead of time by me. The labels on various links were shown in Figure 3.3.

Lengths of Labeled Links

The length of a labeled link is equal to the *intrinsic link-length* of its label node (e.g., *opposite*) if the label node is not fully active, and is equal to the *shrunk link-length* of its label node if the label is fully active.

The intrinsic link-lengths assigned to the various label nodes are:

predecessor: 60

successor: 60

sameness: 0

identity: 0

opposite: 80

The shrunk link-length for each label node is .4 times the intrinsic link-length.

Lengths of Fixed-Length Links

For all links from nodes to their superordinate category-type nodes (e.g., *A* → *letter-category*, or *left* → *direction*) the length is the difference in conceptual depth between the two nodes. That is, the closer they are in conceptual depth, the shorter the link.

For other fixed-length links, the lengths are set by hand. The values are:

A → *first*: 75

Z → *last*: 75

letter-category → *number-category*: 95

number-category → *letter-category*: 95

letter → *group*: 90

group → *letter*: 90

predecessor → *predecessor-group*: 60

successor → *successor-group*: 60

sameness → *sameness group*: 30

predecessor-group → *predecessor*: 90

successor-group → *successor*: 90

sameness-group → *sameness*: 90

single → *whole*: 90

whole → *single*: 90

left → *leftmost*: 90

leftmost → *left*: 90

right → *rightmost*: 90

rightmost → *right*: 90

successor-group → *number-category*: 95

predecessor-group → *number-category*: 95

sameness-group → *number-category*: 95

sameness-group → *letter-category*: 50

At present, all types of groups can have length (number-category) descriptions, whereas only sameness groups can have letter-category descriptions. This is why there are no *successor-group* → *letter-category* or *predecessor-group* → *letter-category* links.

In addition to the links listed above, it was necessary to add certain links for the purpose of making certain concept-mappings compatible and internally coherent (e.g., *first* ⇒ *last* should support *leftmost* ⇒ *rightmost*), but to disallow spreading activation over these links. All of the following links have a fixed length of 100, which means that no activation spreads over them, even though the program considers the nodes to be related for the purpose of calculating the strengths of correspondences. This mechanism is not ideal, and should probably be modified in future work on Copycat.

right → *leftmost*: 100

leftmost → *right*: 100

left → *rightmost*: 100

rightmost → *left*: 100

leftmost → *first*: 100

first → *leftmost*: 100

rightmost → *first*: 100

first → *rightmost*: 100

leftmost → *last*: 100

last → *leftmost*: 100

rightmost → *last*: 100

last → *rightmost*: 100

B.2 Other Slipnet Parameters

Number of codelets run before a Slipnet update: 15

Number of Slipnet updates for initially-clamped nodes (i.e., *letter-category* and *string-position*) to be clamped: 50

B.3 Slipnet Formulas

Activation decay: Each node loses (100 – conceptual-depth) percent of its activation at each Slipnet update.

Activation spread: If a node is fully active, it spreads activation to each of the nodes it is linked to. Each neighboring node gets *link-length* percent of the original node's activation. In the current version of the program, the program always uses the intrinsic link-length, rather than the shrunk link-length, for this calculation, even when the label node for this link is active. Shrunk link-lengths are used only by codelets in evaluating slippages, bonds, etc. When I used shrunk link-length for spreading activation, the network tended to become too active. It is possible that a different mechanism (e.g., some kind of inhibition technique) should be used to control activation in the network—this is a topic for future work on Copycat.

B.4 Temperature Formulas

The temperature is updated along with the Slipnet, every 15 codelet steps. The formula for calculating the temperature is

(.8 * [the weighted average of the unhappiness of all objects, weighted by their relative importance]) + (.2 * [100 – strength(rule)])

The factors .8 and .2 are the weights given to the two components (the unhappinesses of objects in the Workspace and the inverse of the strength of the rule) in this calculation. As discussed in Chapter 4, there are some problems with this weighting scheme, which result in implausible temperature values for some answers.

As was discussed in Chapter 3, in addition to affecting the choice of which codelet to run next (how this is implemented is described in the next section), temperature affects several probabilistic choices made by codelets. This is implemented as follows.

Before a codelet makes a probabilistic choice based on probability p (e.g., the probability whether to post a follow-up codelet to test the strength of a given structure), it adjusts p (a number between 0 and 1 according to the current temperature by sending it through a filter. The filter adjusts probabilities lower than .5 up and probabilities higher than .5 down by an amount that depends on the temperature (the higher the temperature, the closer probabilities are brought to .5). The filter is at present a fairly complicated formula,

arrived at partially by trial and error. It gives the desired numbers, but it is inelegant. The whole formula should eventually be simplified. The formula, written in Common Lisp, is given below (*temperature* is the global temperature variable).

```
(defun adjust-probability (p)
    (cond ((= p 0) 0)
          ((≤ p .5)
           (let* ((term1 (max 1 (truncate (abs (log p 10)))))
                  (term2 (expt 10 (- (- term1 1))))
                 (min .5 (+ p (* (/ (- 10 (sqrt (- 100 *temperature*))) 100)
                                 (- term2 p)))))))
          ((> p .5)
           (max .5 (- 1 (+ (- 1 p)
                           (* (/ (- 10 (sqrt (- 100 *temperature*))) 100)
                              (- 1 (- 1 p)))))))))
```

Temperature also affects the degree of randomness with which fights between competing structures are decided. A fight is decided probabilistically based on the respective strengths of the structures involved, but these strengths are first sent through a filter that adjusts them according to the current temperature, enhancing differences in strengths more and more as the temperature falls. The filter is:

$$\text{adjusted-strength} = \text{strength}^{\frac{100-\text{temperature}}{30}+.5}$$

The constants 30 and .5 are for scaling purposes and were determined by trial and error.

B.5 Coderack Parameters and Formulas

The following describes how the temperature-controlled probabilistic choice of codelets works. There are a fixed number of possible urgencies that can be assigned to codelets (currently 7), and each of those urgency "bins" is given a new value each time the temperature is updated. The bins are numbered 1, 2,...,*highest-bin-number*. The function for the value of each bin is:

$$\text{urgency} = \text{bin-number}^{\frac{(100-\text{temperature})+10}{15}}$$

The constants 10 and 15 are for scaling purposes and were determined by trial and error. The calculated urgency values are then used to directly make a probabilistic choice of the next codelet to run.

The coderack is limited to a certain size (currently 100), and if this limit is exceeded by new codelets being posted, codelets are chosen probabilistically (as a function of their urgency and their age on the coderack) to be deleted until the limit is again reached. The following value is first assigned to each codelet c in the Coderack:

$$\text{age}(c) * [\text{highest-urgency} - \text{urgency}(c)],$$

and then these values are converted into probabilities that are used to decide which codelets to delete. Thus, the older the codelet and the lower its urgency, the more likely it will be chosen to be deleted.

APPENDIX C

More Detailed Descriptions of Codelet Types

This appendix describes the various codelet types in more detail than was given in Chapter 3. The various probability values detailed in these descriptions are the values *before* the adjustment for temperature (described in Appendix B) is applied.

Description-Building Codelets

Bottom-up description-scout (no arguments):

1. Choose an object in the Workspace probabilistically as a function of salience.
2. Choose a relevant description of the object probabilistically as a function of the activation of the descriptors.
3. See if this descriptor has any "has property" links in the Slipnet that are short enough (whether they are short enough is decided probabilistically with probability equal to ((100 – link-length) / 100).
4. If not, then fizzle. Otherwise, choose one of the close-enough properties probabilistically as a function of degree of association and activation.
5. Propose a description of the object, based on this property, and post a description-strength-tester codelet whose urgency is a function of the activation of the description-type (e.g., if the proposed descriptor is *first*, then the description-type is *alphabetic-position*).

Top-down description-scout (argument: a description-type node):

1. Choose an object in the Workspace probabilistically as a function of salience.
2. Test all the possible descriptors of the given description type to see if any can be applied to this object (e.g., if the description-type is *alphabetic-position*, then *first* and *last* will be tested).

3. If no descriptors of this type can be applied to the chosen object, then fizzle. Otherwise, choose one of the applicable descriptors probabilistically (as a function of activation) and post a description-strength-tester codelet whose urgency is a function of the activation of the description-type.

Description-strength-tester (argument: a proposed description):

1. Activate the proposed descriptor in the Slipnet (i.e., give it full activation).
2. Calculate the proposed description's strength.
3. Decide probabilistically whether or not to continue, as a function of the proposed description's strength. If no, then fizzle. Otherwise, post a description-builder codelet whose urgency is a function of the proposed description's strength.

Description-builder (argument: a proposed description):

1. If this description is already attached to the given object, then fizzle. Otherwise build the description, and activate the descriptor and the description-type in the Slipnet.

Bond-Building Codelets

Bottom-up bond-scout (no arguments):

1. Choose an object in the Workspace probabilistically as a function of salience.
2. Choose an adjacent object probabilistically as a function of salience.
3. Choose a "bonding-facet" (i.e., what aspect of the objects to look at in making a bond—at present the only possible bonding-facets are *letter-category* and *number-category*) probabilistically as a function of the possible facets' local support (i.e., a function of how many objects in the string have this type of description) and activation.
4. See if each chosen object has a descriptor of the given bonding-facet (e.g., *letter-category*). If not, then fizzle.
5. If so, then see if there is a relationship in the Slipnet between these two descriptors. If not, then fizzle.
6. If so, then propose a bond between these two objects, and post a bond-strength-tester codelet whose urgency is a function of the proximity of the two descriptors in the Slipnet.

Top-down bond-scout: category (argument: a bond-category node):

1. Choose a string to work in probabilistically, as a function of both the support of the given bond-category in each string (e.g., if the bond-category is *successor*, then the string with more successor bonds is more likely to be chosen) and the average unhappiness of objects in the string (the string with more unhappy objects is more likely to be chosen).
2. Choose an object in the Workspace probabilistically as a function of salience.
3. Choose an adjacent object probabilistically as a function of salience.
4. Choose a bonding-facet.
5. See if each chosen object has a descriptor of the given bonding-facet (e.g., *letter-category*). If not, then fizzle.
6. If so, then see if there is a link in the Slipnet of the given category between these two descriptors. If not, then fizzle.
7. If so, then propose a bond between these two objects, and post a bond-strength-tester codelet whose urgency is a function of the proximity of the two descriptors in the Slipnet.

Top-down bond-scout: direction (argument: a direction node):

1. Choose a string to work in probabilistically, as a function of both the support of the given direction in each string (e.g., if the direction is *right*, then the string with more right-going bonds is more likely to be chosen) and the average unhappiness of objects in the string (the string with more unhappy objects is more likely to be chosen).
2. Choose an object in the Workspace probabilistically as a function of salience.
3. Choose an adjacent object in the given direction.
4. Choose a bonding-facet.
5. See if each chosen object has a descriptor of the given bonding-facet (e.g., *letter-category*). If not, then fizzle.
6. If so, then see if there is some link in the Slipnet between these two descriptors (in the given direction). If not, then fizzle.

7. If so, then propose a bond between these two objects, and post a bond-strength-tester codelet whose urgency is a function of the proximity of the two descriptors in the Slipnet.

Bond-strength-tester (argument: a proposed bond):

1. Calculate the proposed bond's strength.
2. Decide probabilistically whether or not to continue, as a function of the proposed bond's strength. If no, then fizzle. Otherwise, post a bond-builder codelet whose urgency is a function of the proposed bond's strength.
3. Activate (in the Slipnet) the two descriptors being related, and the bonding-facet (e.g., *letter-category*).

Bond-builder (argument: a proposed bond):

1. If the same bond has already been built between the two objects, then fizzle.
2. Otherwise, fight with any incompatible bonds, groups, and correspondences. If any fight is lost, then fizzle. Otherwise, break all incompatible structures, build the proposed bond, and activate (in the Slipnet) the new bond's bond-category and direction.

Group-Building Codelets

Note that any time a new group is proposed, the proposed group is automatically given a number of descriptions, including a *group-category* description, an *object-category* description (i.e., *group*), a *letter-category* description if it is a sameness group, and a *string-position* description if applicable. There is also some probability that it will be given a *length* description. This probability is a function of the length of the group (the shorter, the more likely) and the activation of *number-category* (the higher, the more likely).

Top-down group-scout: category (argument: a group-category node):

1. Choose one of the strings probabilistically as a function of support for the bond-category associated with the given group-category (e.g., if the group-category is *successor-group*, then the string with more successor bonds is more likely to be chosen) and the average unhappiness of objects in the string (the string with more unhappy objects is more likely to be chosen).

2. Choose an object in the Workspace probabilistically as a function of salience.
3. Choose a window in the string (with the chosen object at one end) in which to look for adjacent bonds of the given category and all in the same direction (no matter which one, as long as they are all the same). The choice is probabilistic, with larger windows being more likely to be chosen.
4. Choose a scanning direction for scanning the window.
5. Start from the chosen object, and scan through the chosen window in the chosen scanning direction until no more adjacent bonds of the given category are found. (If no bonds are found, then decide probabilistically whether or not to propose a single-letter group, as a function of local support in the string for the given group-category, and the activation of *number-category*.)
6. If no bonds are found (and a single-letter group is not being proposed), then fizzle.
7. Otherwise, propose a group based on the bonds found, and post a group-strength-tester codelet whose urgency is a function of the proximity encoded by the group's bond-category (e.g., *successor*, if the proposed group is based on successor bonds).

Top-down group-scout: direction (argument: a direction node):

1. Choose one of the strings probabilistically as a function of support for the given direction (e.g., if the direction is *right*, then the string with more right-going bonds is more likely to be chosen) and the average unhappiness of objects in the string (the string with more unhappy objects is more likely to be chosen).
2. Choose an object in the Workspace probabilistically as a function of salience.
3. Choose a window in the string in which to look for adjacent bonds of the given direction and all of the same category (no matter which one, as long as they are all the same). The choice is probabilistic, with larger windows being more likely to be chosen.
4. Choose a scanning direction for scanning the window.
5. Start from the chosen object, and scan through the chosen window in the chosen scanning direction until no more adjacent bonds of the given direction are found.
6. If no bonds are found, then fizzle.

7. Otherwise, propose a group based on the bonds found, and post a group-strength-tester codelet whose urgency is a function of the proximity encoded by the group's bond-category (e.g., *successor*, if the proposed group is based on successor bonds).

Group-string-scout (no arguments):

1. Choose a string at random.
2. See if there is a set of adjacent bonds of the same type and direction spanning the string.
3. If not, then fizzle. Otherwise, propose a group based on these bonds, and post a group-strength-tester codelet whose urgency is a function of the proximity encoded by the group's bond-category.

Group-strength-tester (argument: a proposed group):

1. Calculate the proposed group's strength.
2. Decide probabilistically whether or not to continue, as a function of the proposed group's strength. If no, then fizzle. Otherwise, post a group-builder codelet whose urgency is a function of the proposed group's strength.
3. Activate (in the Slipnet) the group's bond-category and direction.

Group-builder (argument: a proposed group):

1. If the same group already exists, then fizzle.
2. Otherwise, fight with any incompatible bonds, groups, and correspondences. If any fight is lost, then fizzle. Otherwise, break all incompatible structures, build the proposed group, and activate (in the Slipnet) all the descriptions given to the new group, including the new group's group-category.

Correspondence-Building Codelets

Bottom-up correspondence-scout (no arguments):

1. Choose two objects, one from the initial string and one from the target string, probabilistically as a function of salience.

2. Take all the relevant descriptions of each object and make a list of all possible concept-mappings between the descriptors. A concept-mapping is possible if the two descriptors are identical or if they are linked in the Slipnet by a lateral slippage link (see Section 3.3 for a discussion of the different types of links in the Slipnet).
3. See if there is any concept-mapping on the list that warrants proposing a correspondence between the two objects. Such a concept-mapping has to consist of *distinguishing* descriptors (e.g., the concept-mapping *letter* ⇒ *letter* does not warrant a correspondence on its own) and has to represent a "close-enough" relationship in the Slipnet. Identity mappings are always considered to be close-enough, so if there is a distinguishing identity mapping (i.e., a mapping with distinguishing descriptors, such as *rightmost* ⇒ *rightmost*), then it is a sufficient basis for a correspondence. For slippages, the probability of being considered close-enough is a function of both link-length and conceptual depth of the descriptors. The shorter the link, the more likely it is for the the concept-mapping to be judged close-enough, and, as discussed in Chapter 3, the deeper the descriptors, the more resistance to slippage, so the less likely it is for them to be considered close-enough for a slippage to be made.
4. If there is no concept-mapping that warrants proposing a correspondence, then fizzle. Otherwise, propose a correspondence with *all* the possible concept-mappings that were found in step 2. Once one concept-mapping (e.g., *rightmost* ⇒ *rightmost*) has been determined to be sufficient, then all the others (e.g., *letter* ⇒ *letter*) come along for the ride. Post a correspondence-strength-tester codelet whose urgency is a function of the strengths of the proposed correspondence's *distinguishing* concept-mappings.

Important-object correspondence-scout (no arguments):

1. Choose an object from the initial string probabilistically as a function of importance.
2. Choose a descriptor (from that object's relevant descriptions) probabilistically as a function of conceptual depth (e.g., it might be the descriptor *rightmost*).
3. Try to find an object in the target string with the same descriptor, possibly taking into account a slippage that has already been made (e.g., if the chosen

descriptor is *rightmost*, and if the slippage *leftmost* ⇒ *rightmost* has already been made, then that implies the slippage *rightmost* ⇒ *leftmost*, so this codelet will look for the *leftmost* object rather than the *rightmost* object).

4. If no object in the target string with that descriptor is found then fizzle. Otherwise, proceed as in steps 2–4 of the bottom-up-correspondence-scout to propose a correspondence.

Correspondence-strength-tester (argument: a proposed correspondence):

1. Calculate the proposed correspondence's strength.
2. Decide probabilistically whether or not to continue, as a function of the proposed correspondence's strength. If no, then fizzle. Otherwise, post a correspondence-builder codelet whose urgency is a function of the proposed correspondence's strength.
3. Activate (in the Slipnet) the description-types and descriptors of all of the proposed correspondence's concept-mappings.

Correspondence-builder (argument: a proposed correspondence):

1. If the same correspondence has already been built, then fizzle.
2. Otherwise, fight with any incompatible bonds, groups, and correspondences, and the rule, if an incompatible one has been built. If any fight is lost, then fizzle. Otherwise, break all incompatible structures, build the proposed correspondence, and activate (in the Slipnet) the nodes representing the labels of any slippage in the new correspondence's concept-mappings (e.g., if one of the concept-mappings is *rightmost* ⇒ *leftmost*, then activate *opposite*).

Rule-Building Codelets

Rule-scout (no arguments):

1. Find the letter in the initial string that has been changed, that is, whose replacement in the modified string does not have the same letter-category (the program assumes that exactly one letter will have changed).
2. Get a list of the possible descriptors of the changed letter that can be used in filling in the rule template. These descriptors have to be taken from relevant and

distinguishing descriptions, but there are sometimes some other restrictions as well. If a correspondence has been built from this letter to an object in the target string, then the possible descriptors have to be part of the concept-mappings underlying the correspondence. For example, in "**abc ⇒ abd, ijk ⇒ ?**", if a correspondence has been built from the **c** to the **k**, then this codelet would not propose the rule "Replace *C* by *D*", since the descriptor *C* is not part of that correspondence. Likewise, in "**abc ⇒ abd, xcg ⇒ ?**", if a correspondence has been built between the two **c**'s, then this codelet would not propose the rule "Replace rightmost letter by successor", since the descriptor *rightmost* is not part of that correspondence. If there is no correspondence attached to the changed letter, then all the relevant, distinguishing descriptors are eligible.

3. Choose a descriptor from the list of eligible descriptors probabilistically, as a function of conceptual depth.
4. Choose a descriptor of the letter in the modified string corresponding to the changed letter in the initial string. The choice is also made probabilistically, as a function of conceptual depth. When a replacement structure has been built between the initial-string letter and the modified-string letter, the modified-string letter is given a description corresponding to the relationship between the two letters if there is one. For example, for **abc ⇒ abd**, the **d** would be given the description "successor of the **c**", but for **abc ⇒ abq**, no such description would be given, since there is no relationship in the Slipnet between *C* and *Q*.
5. Propose a rule with the two chosen descriptors, and post a rule-strength-tester codelet whose urgency is a function of the conceptual depth of the two descriptors.

Rule-strength-tester (argument: a proposed rule):

1. Calculate the proposed rule's strength.
2. Decide probabilistically whether or not to continue, as a function of the proposed rule's strength. If no, then fizzle. Otherwise, post a rule-builder codelet whose urgency is a function of the proposed rule's strength.

Rule-builder (argument: a proposed rule):

1. If the proposed rule already exists, then fizzle.

2. Otherwise, if there is a different existing rule, fight with it. If the fight is lost, then fizzle. Otherwise, break the incompatible rule, and build the proposed rule, and activate (in the Slipnet) the two descriptors making up the rule.

Rule-translator (no arguments):

1. Decide whether the temperature is too high to translate the rule. To do this, choose a threshold probabilistically as a function of the amount of structure that has been built so far (this is described in Section 3.4.3), and see if the temperature is above the chosen threshold. If so, fizzle.
2. Otherwise, construct a translated rule by applying the slippages that have been made in the various correspondences to the descriptors in the original rule. Once the translated rule has been built, the program will stop running codelets, and will produce an answer by applying the translated rule to the target string.

Other Codelets

Replacement-finder (no arguments):

1. Choose a letter at random in the initial string. If this letter already has a replacement structure attached to it, then fizzle.
2. Otherwise, get the letter in the corresponding position in the modified string.
3. Build a replacement structure between the two letters.
4. If the two letters have different letter-categories, then if their letter-categories are related in the Slipnet, add a description to the modified-string letter describing the relation (e.g., "successor of the c").

Breaker (no arguments):

1. Decide probabilistically, as a function of temperature, whether or not to fizzle immediately (the lower the temperature, the more likely this codelet is to decide to fizzle).
2. If the decision was made to continue, choose a structure at random. Decide probabilistically, as a function of the structure's strength, whether or not to break it (the weaker it is, the more likely it is to be broken). If the decision is made to break the structure, then break it.

APPENDIX D

Results of Further Experiments on People

D.1 Introduction

Several experiments on people have been suggested to give evidence for Copycat's psychological plausibility, and it is worth discussing which ones I feel are useful, and which not. One suggestion has been to measure precisely the relative times it takes the program to solve various problems and to compare them with people's relative times on the same problems. This is clearly too fine-grained a comparison, since Copycat is not and was never meant to be a model of how people read and process letter-strings. Another suggested experiment is to test whether or not the frequencies of Copycat's different answers for each problem match the frequencies given by a group of human subjects. The suggestion is that, for example, if, given "**abc ⇒ abd, kji ⇒ ?**", 6 out of 10 people answer **kjj**, 3 answer **kjh** and 1 answers **lji**, then the program should be judged on how well it matches these frequencies. This would not be a useful experiment: Copycat is not meant to be a model of how a *population* of people responds to these analogy problems; it is closer to being a model of an individual person, with high-level preferences emerging statistically from micro-biases.

The answer frequencies and temperatures displayed in a bar graph represent the different degrees to which various answers are obvious and preferable to the program. They are meant to correspond to the degree to which an individual would feel a certain answer was obvious or good. For example, in "**abc ⇒ abd, ijk ⇒ ?**", Copycat's more than 50-to-1 ratio of frequencies of answer **ijl** over **ijd** is meant to model the vast difference in immediacy of these two answers in a single person's mind: although the route to **ijd** is always open, it is quite unlikely and is almost never followed. Likewise, the average final temperatures are supposed to represent how an individual, with individual biases, would rate the different answers (though, as was discussed previously, there are some problems

with the way temperature is currently calculated in Copycat). Thus, matching Copycat's answer frequencies against those of a *population* is not the right experiment to do; it would be better to see if Copycat's micro-biases could be tuned to match the behavior of a single person. However, at this point even this experiment is too fine-grained, since there are so many factors that go into a person's answers that involve knowledge that Copycat lacks, and it is also unclear how such a comparison would be done, since it is hard to know, for example, how likely the path to an answer like **ijd** would be in a single person's mind.

At this point, therefore, only limited direct comparisons can be made with people in order to lend more plausibility to the model. The comparisons I felt would be useful and chose to do are the following:

1. *Comparing the range of answers given by the program and by people.* The results of these comparisons were discussed in Chapters 4 and 5.

2. *Comparing the effects (at a coarse-grained level) on the program and on people of small variations in pressures.* The results of these comparisons were discussed in Chapter 5.

3. *Comparing preferences on answers.* If people generally agree that, given the restrictions of the microworld, a given answer is very strong or very weak, then Copycat should concur with this judgment, in the sense of having a low or high average final temperature for that answer. This would lend some plausibility to the program's mechanisms for judging the quality of various ways of perceiving situations. However, if the quality of a given answer is controversial among people, then a comparison with Copycat's judgment cannot be made, since the program is in effect exhibiting its own "taste" in the matter (a result of a large number of micro-biases).

4. *Comparing relative difficulties on different problems.* If people generally find certain problems much more difficult than others (e.g., if "**abc ⇒ abd, xyz ⇒ ?**" is more difficult than "**abc ⇒ abd, ijk ⇒ ?**"), the program should experience the same relative difficulties: it should take longer to solve problems difficult for people than problems simple for people, and if there are any specific difficulties people reliably run into (e.g., in "**abc ⇒ abd, xyz ⇒ ?**", trying to take the successor of *Z* and hitting an impasse), the program should run into them as well. This would lend some plausibility to the program's underlying mechanisms as models of similar mechanisms in humans.

D.2 Comparing Preferences on Answers

For this comparison I used 19 paid subjects who were already familiar with the letter-string domain, having just participated in either the survey described in Chapters 4 and 5 or the timing experiment described in the next section. The subjects were given a list of problems (including the five target problems or slight variants) along with a set of possible answers to each problem with a detailed written justification for each answer (I don't repeat the justifications for each answer here, but they should be clear from earlier discussions). The subjects were asked, for each problem, to rate each given answer on a scale from 5 to 1, with the following adjectives associated with each number: "intelligent" (5), "reasonable" (4), "barely reasonable" (3), "weak" (2), and "stupid" (1). My goal was to see, when there was more or less general agreement among people on the strength or weakness of a certain answer, whether or not Copycat also judged that answer to be strong or weak, as measured by its average final temperature for that answer.

The interesting statistics here for a given answer are the mean rating, which indicates how well people liked the answer on average, and the standard deviation, which indicates how much agreement there was on the rating for that answer. A standard deviation of roughly 1 or less indicates a reasonable amount of agreement on the rating of the answer.

abc ⇒ abd, pqrs ⇒ ?: This is a slight variant of "**abc ⇒ abd, ijk ⇒ ?**". Copycat's performance on this problem is no different from its performance on the original. Four possible answers were given for people to rate, and their ratings were:

Answer	*Mean Rating*	*Std. Dev.*
pqrt	4.6	.5
pqrd	3.2	1.3
pqrs	2.8	1.2
oqrs	1.9	.9

Copycat's average final temperatures for these answers (1000 runs) are as follows:

Answer	*Average Final Temperature*
pqrt	20
pqrd	27
pqrs	60

Again, the point here is not to make a fine-grained comparison between people and Copycat, but rather to see whether, when there is general agreement among people, Copycat

agrees as well. The answer **pqrt** is a clear winner here among people, with a high mean rating (between "reasonable" and "intelligent") and a low standard deviation, indicating general agreement on its merits. This answer is also rated highly by Copycat, with an average temperature of 20. People also fairly reliably rate **oqrs** (simply "Replace leftmost letter by predecessor" with no other justification given) as "weak" or "stupid". In 1000 runs, Copycat never answered **oqrs** (though it is theoretically capable of doing so) since, in the context of this problem, the *rightmost* ⇒ *leftmost* correspondence is extremely weak and unlikely to be made. The other two answers are generally seen by people as weak or barely reasonable, but these ratings are a bit more controversial. Like people, Copycat views **pqrs** as quite weak. Copycat's temperature on **pqrd** is fairly low, which, as was discussed in Chapter 4, reflects a problem with the way temperature is calculated in the program: it does not sufficiently take into account the weakness of a rule like "Replace rightmost letter by *D*".

abc ⇒ abd, iijjkk ⇒ ?: (This was actually given as "**abc ⇒ abd, nnoopp ⇒ ?**", but for consistency's sake, here I use the letters **i**, **j**, and **k**). Four possible answers were given for people to rate, and their ratings were:

Answer	*Mean Rating*	*Std. Dev.*
iijjll	4.1	1.0
iijjkl	3.7	1.3
iijjkd	3.0	2.1
iijjdd	2.8	1.2

Copycat's average final temperatures for these answers (1000 runs) are as follows:

Answer	*Average Final Temperature*
iijjll	28
iijjkl	47
iijjkd	62
iijjdd	41

Here, people more or less agreed that **iijjll** is a good answer, with a mean rating of "reasonable", and a standard deviation of 1. Copycat is in agreement with that assessment: this answer has by far the lowest average temperature. There was less agreement among people on the other answers, though it was agreed to some extent that **iijjkl** is better than the two *D* answers. Copycat ranks **iijjdd** higher than **iijjkl** (although it gets the latter

much more often), since the former reflects a correspondence between the **c** and the *group* **kk**, which is stronger than a *letter* ⇒ *letter* correspondence. This, though, once again shows a flaw in the way Copycat's temperature is calculated: the weakness of the rule "Replace rightmost letter [or group] by *D*" should affect the temperature more than it does.

abc ⇒ abd, kjih ⇒ ?: This is a slight variant of "**abc ⇒ abd, kji ⇒ ?**". Copycat's performance on this problem is basically no different from its performance on the original. Four possible answers were given for people to rate, and their ratings were:

Answer	*Mean Rating*	*Std. Dev.*
kjii	3.9	1.1
ljih	3.6	1.3
kjig	3.4	1.3
kjid	2.9	1.2

Copycat's average final temperatures for these answers (1000 runs) are as follows:

Answer	*Average Final Temperature*
kjii	48
ljih	22
kjig	17
kjid	34

People on average judged **kjii** as a reasonable answer, with a fairly low standard deviation: 14 out of 19 people judged it as "reasonable" or "intelligent". Copycat disagrees with this rating; this answer has a fairly high average temperature because it does not take into account the *structure* of the target string **kjih**. The program much prefers **ljih** and **kjig**. People were more divided on these two answers; on each, more than half the subjects judged it as "reasonable" or "intelligent", but each got a number of low ratings as well.

abc ⇒ abd, mrrjjj ⇒ ?: Three possible answers were given for people to rate, and their ratings were:

Answer	*Mean Rating*	*Std. Dev.*
mrrkkk	4.2	.9
mrrjjk	3.6	1.3
mrrjjjj	3.3	1.5

Copycat's average final temperatures for these answers (1000 runs) are as follows:

Answer	*Average Final Temperature*
mrrkkk	43
mrrjjk	50
mrrjjjj	20

Here people more or less agree that **mrrkkk** is a good answer, with a mean rating of "reasonable" and a standard deviation of .9. Copycat does not agree with this assessment; its average final temperature on this answer is fairly high, since this answer reflects the fact that the program was not able to form a coherent structure out of the the target string. The other answers are more controversial, all having large standard deviations. Copycat's favorite answer by far, **mrrjjjj**, had a mean rating of 3.3 by people with a large standard deviation of 1.5, indicating that it was the most controversial answer. Out of 19 subjects, 10 thought it was reasonable or intelligent, 8 thought it was barely reasonable, weak, or stupid, and one rated it between "barely reasonable" and "reasonable".

abc ⇒ abd, xyz ⇒ ?: Four possible answers were given for people to rate (they were reminded that **xya** was not allowed) and their ratings were:

Answer	*Mean Rating*	*Std. Dev.*
xyd	3.2	1.3
xyz	3.0	1.2
wyz	2.9	1.6
yyz	2.7	1

Copycat's average final temperatures for these answers (1000 runs) are as follows:

Answer	*Average Final Temperature*
xyd	22
xyz	74
wyz	14
yyz	44

Here, people more or less agreed that **yyz** is a fairly weak answer (Copycat also rates it as fairly weak, with a temperature of 44), but otherwise there was not much agreement among the subjects. Again, the most controversial answer is Copycat's favorite, **wyz**. It got the highest number of "intelligent" ratings (5) of all answers to this problem, but it also got the highest number of "stupid" ratings (5). Out of 19 subjects, 7 thought it was

reasonable or intelligent, 11 thought it was barely reasonable, weak, or stupid, and one rated it between "barely reasonable" and "reasonable".

In summary, I think it is difficult to draw any strong conclusions from the overall results of these comparisons of answer-ratings over the five target problems. The main reason is that there was a good deal of controversy on most of the answers (there were only three answers in the entire study whose ratings had standard deviations less than 1). This is to be expected, since one of the reasons for choosing these five problems was the fact that each *does* have a number of different plausible answers, and our goal is for Copycat to have the flexibility necessary to get different answers.

On the few answers where there was a fairly clear consensus among people as to the strength or weakness of one answer relative to the others, Copycat agreed, except in the case of **mrrkkk**, which the people in this study tended to rate high, but which Copycat rates low. Also, as might be expected, the more "creative" answers (e.g., **mrrjjjj**, **wyz**) were also among the most controversial; some people liked them very much, whereas others thought that they were too farfetched.

Another problem is that Copycat's calculation of temperature is imperfect, leading to implausibly low final temperatures on answers such as **ijd** or **iijjdd**.

In order to draw stronger conclusions, it would be necessary to make such comparisons over a wider range of problems (involving more problems with clear-cut "best" answers) and involving more subjects. An interesting, more detailed comparison would be to see if the model's micro-biases could be tuned so that the program's performance matched the different tastes and styles of individuals. However, the program is currently not at the level at which such a fine-grained comparison could be made.

D.3 Comparing Relative Difficulties on Different Problems

For this comparison I used 14 paid subjects, who were each given a verbal description of the letter-string domain and its limitations. They were then asked to solve a set of eight problems that appeared one by one on a computer screen. The first three problems were for training purposes, so that the subjects could get used to the experimental setup, and the next five problems were the five target problems (or slight variants) in random order (different orderings for different subjects). The subjects were timed on how long it took them to give an answer to each problem (though I did not tell them they were being timed because I did not want them to feel any time pressure). The purpose here was to see if the

order of difficulty of the five problems (judged by the amount of time taken to solve them) was the same for the program as for people. After solving each problem, each subject gave a short verbal report on how they thought they solved it.

Unfortunately, a number of factors make it very difficult to compare the results here with those of the program. First, the time taken by Copycat to solve a given problem depends very much on what the final answer is. For example, on the problem "**abc ⇒ abd, mrrjjj ⇒ ?**", the program takes an average of 705 codelet steps to reach answer **mrrkkk** versus 1332 for answer **mrrjjjj**; the latter is a harder answer to reach. So any useful comparison with people would have to involve an answer-by-answer time comparison, but the number of subjects here was too small to get the range of answers and number of samples needed for a comparison (e.g., only one subject answered **mrrjjjj**, and on some of the problems, a number of subjects gave answers that Copycat cannot get). Another problem is noise in the data: even with the three training problems, a few of the subjects still had trouble using the keyboard correctly, which increased the time recorded for various answers, and again, the number of subjects was too small to overcome the noise problem. My conclusion is that in order to be useful, this experiment would require many more subjects than I was able to run, and certain design problems would have to be corrected (e.g., more training problems should be used in order for subjects to get used to the experimental setup). Therefore, the results given here should be considered to be those of a pilot study rather than those of a full-fledged experiment.

The comparisons of average overall time for each problem are given below, with the caveat that I don't think that these results are very meaningful. In any case, what is to be compared here is the time-ranked order of the five problems, and any significant differences in time between different problems within a set. The times for the human subjects are given in average number of seconds, and the times for Copycat are given in average number of codelets run; these numbers cannot be directly compared in any way.

The times for the five problems solved by the 14 subjects were:

1. "**abc ⇒ abd, ijkl ⇒ ?**" (average time: 24 seconds)

2. "**abc ⇒ abd, eeffgghh ⇒ ?**" (average time: 37 seconds)

3. "**abc ⇒ abd, srqp ⇒ ?**" (average time: 54 seconds)

4. "**abc ⇒ abd, mrrjjj ⇒ ?**" (average time: 55 seconds)

5. "**abc ⇒ abd, xyz ⇒ ?**" (average time: 59 seconds)

The times for these five problems when solved by Copycat are:

1. "**abc ⇒ abd, ijkl ⇒ ?**" (average number of codelets run: 341)

2. "**abc ⇒ abd, eeffgghh ⇒ ?**" (average number of codelets run: 800)

3. "**abc ⇒ abd, srqp ⇒ ?**" (average number of codelets run: 497)

4. "**abc ⇒ abd, mrrjjj ⇒ ?**" (average number of codelets run: 850)

5. "**abc ⇒ abd, xyz ⇒ ?**" (average number of codelets run: 3322)

In spite of the caveat given above, a few interesting points can be made here. As common sense would tell us, the problem "**abc ⇒ abd, ijkl ⇒ ?**" seems to be the easiest for both people and Copycat (and, of course, almost everyone answered **ijkm**), and the problems "**abc ⇒ abd, mrrjjj ⇒ ?**" and "**abc ⇒ abd, xyz ⇒ ?**" seem significantly more difficult for both the subjects and Copycat. On "**abc ⇒ abd, mrrjjj ⇒ ?**", one subject gave a report expressing a sense of pressures similar to those pushing Copycat: "I was pretty lost with this one, since I didn't see any patterns resembling the given example; the letters in the string I was given [i.e., **mrrjjj**] didn't relate to each other in the same way that the others in the given example [i.e., **abc ⇒ abd**] did. The given letters weren't successors in the alphabet." On "**abc ⇒ abd, xyz ⇒ ?**", all 14 subjects reported "hitting the snag"—that is, trying to take the successor of *Z* and failing (as reported by them). Thus it would be implausible if Copycat easily bypassed this snag and went directly to another answer: the program hit this snag on all but 2% of its runs. After hitting the snag, all but one of the subjects proposed the answer **xya** (the other one reported thinking of it, but assumed it would not be allowed). They were told that this answer, while very reasonable, was not possible given the restrictions of the domain, and were then asked to come up with another answer (the time taken to give these instructions was not included in the recorded solution time).

Some of the problems with the model discussed in Chapter 6 have an effect on the timing differences here. The problem "**abc ⇒ abd, eeffgghh ⇒ ?**" takes Copycat significantly longer than "**abc ⇒ abd, ijkl ⇒ ?**" does, whereas the difference for people is not that great. As was discussed in Chapter 6, one reason for this seems to be that once people start to perceive groups in the string, they get the idea very quickly, whereas such top-down

forces in Copycat, although they exist, are still too weak; they don't sufficiently accelerate this view once it begins to be perceived. Likewise, the problem "**abc ⇒ abd, xyz ⇒ ?**" takes Copycat far longer than any other problem, whereas the difference for people is not that great. This is in part due to the fact that people tend to give up fairly quickly when faced with an impasse and give an answer that they may not find totally satisfying. But the large amount of time taken by Copycat on this problem is also due to its loopish behavior: since it lacks appropriate self-watching mechanisms, it gets trapped in the same state again and again, trying to take the successor of Z and failing.

D.4 Summary

Of the four types of comparisons I did, the first two (comparing the range of answers given by people and by Copycat, and comparing the effects on the program and on people of small variations in pressures) were the most useful in showing where the program succeeds and where it is lacking. The other two comparisons were more problematic. The answer-ratings comparison showed that there is a good deal of disagreement among people on the quality of various answers to these five problems, and I don't think any general conclusions can be made about these results. It would be very interesting to see if Copycat could be "tuned" to match the preferences of different individuals on a wide range of problems; this is an experiment that will be left for future work on this project. The comparison of relative difficulty made a few interesting points, discussed above, but had some design problems and also lacked enough subjects to be conclusive. These last two comparisons could obviously be extended, and the way in which they were carried out could be much improved.

BIBLIOGRAPHY

BIBLIOGRAPHY

Anderson, J. R. (1983). *The architecture of cognition.* Cambridge, MA: Harvard University Press.

Barsalou, L. W. (1989). Intraconcept similarity and its implications for interconcept similarity. In S. Vosniadou and A. Ortony (Eds.), *Similarity and analogical reasoning,* 76– 121. Cambridge, England: Cambridge University Press.

Bongard, M. (1970) *Pattern recognition.* Hayden Book Co. (Spartan Books).

Burstein, M. & Adelson, B. (1987). Mapping and integrating partial mental models. In *Proceedings of the Ninth Annual Conference of the Cognitive Science Society.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Chalmers, D. J., French, R. M., & Hofstadter, D. R. (1990). *High-level perception, representation, and analogy.* Unpublished manuscript, Center for Research on Concepts and Cognition, Indiana University, Bloomington, IN.

Collins, A. M. & Loftus, E. F. (1975). A spreading activation theory of semantic memory. *Psychological Review, 82* 407–428.

Cooper, L. (1913). *Aristotle on the art of poetry: Translation of Aristotle's Poetics.* New York: Harcourt, Brace, and Co.

Cornford, F. M. (1935). *Plato's theory of knowledge: The Thaeatetus and the Sophist translated.* New York: Harcourt, Brace, and Co.

deGroot, A. (1965). *Thought and choice in chess.* The Hague: Mouton.

Erman, L. D., Hayes-Roth, F., Lesser, V. R., & Raj Reddy, D. (1980). The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys, 12*(2), 213–253.

Evans, T. G. (1968). A program for the solution of a class of geometric-analogy intelligence-test questions. In M. Minsky (Ed.), *Semantic information processing.* Cambridge, MA: MIT Press.

Falkenhainer, B., Forbus, K. D. & Gentner, D. (1989). The structure-mapping engine. *Artificial Intelligence, 41*(1), 1–63.

Farmer, J. D., Packard, N. H., & Perelson, A. S. (1986). The immune system, adaptation, and machine learning. *Physica D, 22,* 187–204.

Feldman, J. & Ballard, D. (1982). Connectionist models and their properties. *Cognitive Science, 6*(3), 205–254.

Forrest, S. (1990): Introduction. In S. Forrest (Ed.), *Emergent computation.* Cambridge, MA: MIT Press.

French, R. M. & Henry, J. (1988) La traduction en français des jeux linguistiques de *Gödel, Escher, Bach. Mèta, 33*(2), 133–142.

Fromkin V. (Ed.) (1980). *Errors in linguistic performance: Slips of the tongue, ear, pen, and hand.* New York: Academic Press.

Gentner, D. (1983). Structure-mapping: A theoretical framework for analogy. *Cognitive Science, 7*(2).

Gick, M. L. & Holyoak, K. J. (1983). Schema induction and analogical transfer. *Cognitive Psychology, 15*, 1–38.

Hall, R. P. (1989) Computational approaches to analogical reasoning. *Artificial Intelligence 39*, 39–120.

Hinton, G. E. & Sejnowski, T. J. (1986), Learning and relearning in Boltzmann machines. In D. E Rumelhart and J. L. McClelland (Eds.), *Parallel distributed processing*, 282–317. Cambridge, MA: Bradford/MIT Press.

Hofstadter, D. R. (1979). *Gödel, Escher, Bach: an Eternal Golden Braid.* New York: Basic Books.

Hofstadter, D. R. (1982). *The search for essence 'twixt medium and message: What is the essence of an idea?* CRCC Report No. 4, Center for Research on Concepts and Cognition, Indiana University, Bloomington, IN.

Hofstadter, D. R. (1983). The architecture of Jumbo. *Proceedings of the International Machine Learning Workshop.* Monticello, IL.

Hofstadter, D. R. (1984a). *The Copycat project: An experiment in nondeterminism and creative analogies.* AI Memo No. 755, Massachusetts Institute of Technology, Cambridge, MA.

Hofstadter, D. R. (1984b). *Simple and not-so-simple analogies in the Copycat domain.* CRCC Report No. 9, Center for Research on Concepts and Cognition, Indiana University, Bloomington, IN.

Hofstadter, D. R. (1985a). Analogies and roles in human and machine thinking. In *Metamagical themas*, 547–603. New York: Basic Books.

Hofstadter, D. R. (1985b). On the seeming paradox of mechanizing creativity. In *Metamagical themas*, 526–546. New York: Basic Books.

Hofstadter, D. R. (1985c). Variations on a theme as the crux of creativity. In *Metamagical themas*, 232–259. New York: Basic Books.

Hofstadter, D. R. (1985d). Waking up from the Boolean dream: Subcognition as computation. In *Metamagical themas*, 631–665. New York: Basic Books.

Hofstadter, D. R. (1987). *Fluid analogies and human creativity.* CRCC Report No. 16, Center for Research on Concepts and Cognition, Indiana University, Bloomington, IN.

Hofstadter, D. R., Clossman, G., & Meredith, M. J. (1982). *SEEK-WHENCE: A project in pattern understanding.* CRCC Report No. 3, Center for Research on Concepts and Cognition, Indiana University, Bloomington, IN.

Hofstadter, D. R., & Gabora, L. M. (1990) Synopsis of the workshop on humor and cognition. *Humor, 2*(4), 417–440

Hofstadter, D. R., Mitchell, M., & French, R. M. (1987). *Fluid concepts and creative analogies: A theory and its computer implementation.* Technical Report 10, Cognitive Science and Machine Intelligence Laboratory, University of Michigan, Ann Arbor, MI.

Hofstadter, D. R. & Moser, D. J. (1989). To err is human; to study error-making is cognitive science. *Michigan Quarterly Review, 28*(2), 185–215.

Holland, J. H. (1975). *Adaptation in natural and artificial systems.* Ann Arbor, MI: University of Michigan Press.

Holland, J. H. (1986). Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In R. Michalski et al. (Eds.), *Machine learning: An artificial intelligence approach: Vol. 2.* Los Altos, CA: Morgan Kaufmann.

Holland, J. H. (1988) The dynamics of searches directed by genetic algorithms. In Y. C. Lee (Ed.), *Evolution, learning, and cognition.* World Scientific Press.

Holland, J. H., Holyoak, K. J., Nisbett, R. E., & Thagard, P. R. (1986). *Induction.* Cambridge, MA: Bradford/MIT Press.

Holyoak, K. J. (1984). Analogical thinking and human intelligence. In R. J. Sternberg (Ed.), *Advances in the psychology of human intelligence*, Vol. 2, 199–230. Hillsdale, NJ: Lawrence Erlbaum Associates.

Holyoak, K. J. & Thagard, P. (1989). Analogical mapping by constraint satisfaction. *Cognitive Science, 13*(3), 295– 355.

Johnson-Laird (1989) Analogy and the exercise of creativity. In S. Vosniadou and A. Ortony (Eds.), *Similarity and analogical reasoning*, 313–331. Cambridge, England: Cambridge University Press.

Kahneman, D. & Miller, D. T. (1986). Norm theory: Comparing reality to its alternatives. *Psychological Review, 93*(2), 136– 153.

Kaplan S. & Kaplan, R. (1982). *Cognition and environment.* New York: Praeger Publishers.

Kedar-Cabelli, S. (1988a). Analogy—from a unified perspective. In D. H. Helman (Ed.), *Analogical reasoning*, 65–103. Dordrecht, The Netherlands: Kluwer Academic Publishers.

Kedar-Cabelli, S. (1988b). Towards a computational model of purpose-directed analogy. In A. Prieditis (Ed.), *Analogica.* Los Altos, CA: Morgan Kaufmann.

Kirkpatrick, S., Gelatt Jr., C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science, 220,* 671–680.

Kobsa, A. (1987). What is explained by AI models? In R. Born (Ed.), *Artificial intelligence: The case against.* London: Croom Helm.

Kotovsky, K. & Simon, H. A. (1973) Empirical tests of a theory of human acquisition of concepts for sequential patterns. *Cognitive Psychology, 4,* 399–424.

Lakoff, G. (1987). *Women, fire, and dangerous things.* Chicago: University of Chicago Press.

Lakoff, G. & Johnson, M. (1980). *Metaphors we live by.* Chicago: University of Chicago Press.

Longuet-Higgins, H. C. (1981). Artificial intelligence: A new theoretical psychology. *Cognition, 10,* 197–200.

McClelland, J. L. & Rumelhart, D. E. (1981). An interactive activation model of context effects in letter perception: Part 1. An account of basic findings. *Psychological Review, 88,* 375–407.

McDermott, D. (1981) Artificial intelligence meets natural stupidity. In J. Haugland (Ed.), *Mind design.* Cambridge, MA: MIT Press.

Meehan, J. (1976). *The metanovel: Writing stories by computer.* Technical Report 74, Computer Science Department, Yale University, New Haven, CT.

Meredith, M. J. (1986). *Seek-Whence: A model of pattern perception.* Technical Report No. 214, Computer Science Department, Indiana University, Bloomington, IN.

Moser, D. J. (1988). *If this paper were in Chinese, would Chinese people understand the title?* CRCC Report No. 28, Center for Research on Concepts and Cognition, Indiana University, Bloomington, IN.

Moser, D. J. (1989). *The translation of Gödel, Escher, Bach into Chinese.* CRCC Report No. 31, Center for Research on Concepts and Cognition, Indiana University, Bloomington, IN.

Norman, D. A. (1981). Categorization of action slips. *Psychological Review, 88*(1), 1–15.

Pagels, H. R. (1988). *The dreams of reason.* New York: Simon & Schuster.

Pivar, M. & Finkelstein, M. (1964): Automation, using LISP, of inductive inference on sequences. In E. C. Berkeley and D. Bobrow (Eds.), *The programming language LISP: Its operation and applications,* 125–136. Cambridge, MA: Information International, Inc.

Quillian, M. R. (1968). Semantic memory. In M. Minsky (Ed.), *Semantic information processing.* Cambridge, MA: MIT Press.

Reagan, N. with Novak, W. (1989). *My turn: The memoirs of Nancy Reagan.* New York: Random House.

Rosch, E. & Lloyd, B. B. (1978). *Cognition and categorization.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, (Eds.), *Parallel distributed processing*, 318–362. Cambridge, MA: Bradford/MIT Press,

Rumelhart, D. E. & McClelland, J. L. (Eds.) (1986). *Parallel distributed processing.* Cambridge, MA: Bradford/MIT Press.

Schank, R. C. (1983). *Dynamic memory.* Cambridge, England: Cambridge University Press.

Schank, R. C., & Leake, D. B. (1989). Creativity and learning in a case-based explainer. *Artificial Intelligence, 40*(1–3), 353–385.

Simon, H. A. (1972). Complexity and the representation of patterned sequences of symbols. *Psychological Review, 79*(5), 369–382.

Simon, H. A., & Kotovsky, K. (1963). Human acquisition of concepts for sequential patterns. *Psychological Review, 70*(6), 534–546.

Skorstad, J., Falkenhainer, B., & Gentner, D. (1987). Analogical processing: A simulation and empirical corroboration. In *Proceedings of the American Association for Artificial Intelligence, AAAI-87.* Los Altos, CA: Morgan Kaufmann.

Smith, E. E. & Medin, D. L. (1981). *Categories and concepts.* Cambridge, MA: Harvard University Press.

Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. In D. E. Rumelhart and J. L. McClelland (Eds.), *Parallel distributed processing*, 194–281. Cambridge: Bradford/MIT Press.

Smolensky, P. (1988). On the proper treatment of connectionism. *Behavioral and Brain Sciences, 11*(1), 1–14

Thagard, P. (1989). Explanatory coherence. *Behavioral and Brain Sciences, 12*(3), 435–467.

Thagard, P., Holyoak, K. J., Nelson, G., & Gochfeld, D. (in press). Analog retrieval by constraint satisfaction. *Artificial Intelligence.*

Turner, M. (1988). Categories and analogies. In D. H. Helman (Ed.), *Analogical reasoning*, 3–24. Dordrecht, The Netherlands: Kluwer Academic Publishers.

Yukawa, H. (1973a) Creative thinking in science. In *Creativity and intuition: A physicist looks at east and west.* New York: Kodansha International, Ltd.

Yukawa, H. (1973b) Meson theory in its developments. In *Creativity and intuition: A physicist looks at east and west.* New York: Kodansha International, Ltd.