PARALLEL MODELS OF ASSOCIATIVE MEMORY

Updated Edition

Edited by

Geoffrey E. Hinton

University of Toronto

James A. Anderson

Brown University

LEA LAWRENCE ERLBAUM ASSOCIATES, PUBLISHERS

1989 Hillsdale, New Jersey Hove and London

Comments on Chapter 6

Geoffrey E. Hinton
University of Toronto

This chapter describes one of the first attempts to implement propositional knowledge using distributed representations. It introduces the idea of a "role-specific" representation that combines the identity of an object with its role in some larger structure. Once roles and identities have been conjoined in this way it is easy to implement a compositional representation of the larger structure using a set of role-specific representations of its parts.

The chapter demonstrates that this way of representing structures has some attractive consequences provided appropriate patterns of activity are used. If similar objects are represented by similar patterns we get automatic generalization. In particular, if the units that are active in the representation of a type are a subset of the units that are active in the representation of an instance of that type we get automatic property-inheritance. Derthick (1987) has shown how these ideas can be extended to allow more complex representations that have role-hierarchies. He has also shown that this kind of system can be given a formal semantics (Derthick, 1987). McClelland and Kawamoto (1986) have shown how role-specific representations can be used to implement a connectionist system that maps from surface roles to semantic roles. Their system uses semantic information to decide between alternative possible mappings.

The main weakness of the chapter is that it does not provide a satisfactory method of automatically generating the patterns of activity that act as role-specific representations. The chapter does stress the importance of this problem, and the section on learning mentions an unsupervised learning procedure that is described in detail in Pearlmutter and Hinton (1986). With

the advent of more powerful learning procedures like back-propagation (Rumelhart, Hinton, & Williams, 1986), it became feasible for the network itself to discover appropriate internal representations (Hinton, 1986).

One apparent disadvantage of role-specific representations is that each object must have a different internal representation for each possible role that it can play in a larger structure. It would be very uneconomical to duplicate the hardware across all possible roles, but fortunately this is not necessary. Section 9.1 of the chapter shows that it is possible to share hardware among similar roles provided each individual structure only involves a small number of roles. Each active unit is then coarsely tuned in both identity space and role space, and each binding of an identity to a role is encoded by a set of active units.

Another apparent disadvantage is that there need be nothing in common between different role-specific representations of the same object. So it appears that there are many regularities that will not be captured by this representational scheme. Suppose, for example, that the network learns that Zenon and Jerry are very similar when they are filling the "agent" role. How can it automatically generalize this similarity to the representations of Zenon and Jerry in the patient role? Part of the answer is given in Section 10 of the chapter, which shows how the many role-specific representations of an object could interact with a single, role-independent, canonical representation. If this canonical representation is used to supervise the *learning* of the role-specific representations, it is possible to capture the regularities that exist between the various role-specific representations. This method of capturing the regularities has not yet been implemented, but it promises to be much more economical than the only implemented alternative which is to copy the weights of the canonical representation each time a role-specific representation is required (McClelland, 1986).

The end of the chapter raises the question of how a connectionist network could enforce consistent variable bindings when matching some propositions to an inference schema. At the time the chapter was written this seemed like a very difficult task, but Touretzky and Hinton (1985) have now demonstrated one way of performing it in a connectionist network.

REFERENCES

Derthick, M. A. A connectionist architecture for representing and reasoning about structured knowledge. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, Amherst, Mass., 1987.

Hinton, G. E. Learning distributed representations of concepts. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, Amherst, Mass., 1986.

McClelland, J. L. The programmable blackboard model of reading. In J. L. McClelland, &

D. E. Rumelhart, (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition. Vol. 2: Applications*, Cambridge, Mass.: MIT Press, 1986.

McClelland, J. L. & Kawamoto, A. H. Mechanisms of sentence processing: Assigning roles to constituents of sentences. In J. L. McClelland & D. E. Rumelhart (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition. Vol. 2: Applications.* Cambridge, Mass.: MIT Press, 1986.

Pearlmutter, B. A. & Hinton, G. E. G-maximization: An unsupervised learning procedure for discovering regularities. In J. Denker (Ed.), *Neural networks for computing*. American Institute of Physics, 1986.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. Learning representations by back-propagating errors. *Nature, 1986, 323*, 533–536.

Touretzky, D. S. & Hinton, G. E. Symbols among the neurons: Details of a connnectionist inference architecture. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, Calif., 1985.

6 Implementing Semantic Networks in Parallel Hardware

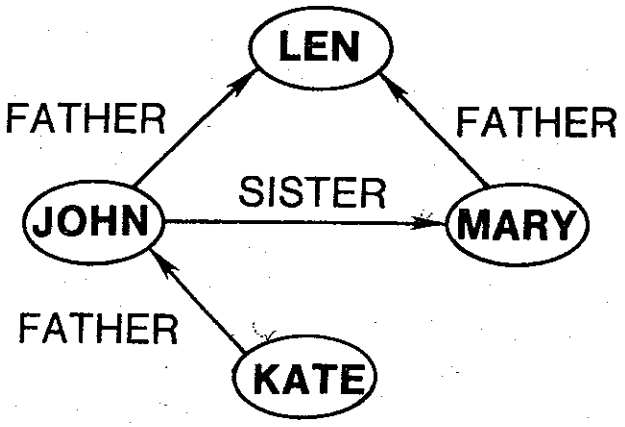
Geoffrey E. Hinton
M.R.C. Applied Psychology Unit
Cambridge, England

6.1. INTRODUCTION

There are two very different ways of implementing semantic networks (see Fig. 6.1) in networks of simple hardware units. The obvious approach is to make different nodes in the semantic net correspond to different hardware units and to make links between semantic nodes correspond to hardware links between units. This type of direct implementation is advocated by Fahlman (Chapter 5, this volume) and Feldman (Chapter 2, this volume). It is also implicit in models that talk about "activation" spreading from one semantic node to another (Collins & Loftus, 1975; Levin, 1976).

A radically different approach is to make each node in the semantic net correspond to a particular pattern of activity on a large assembly of units. Different semantic nodes may then be represented by different patterns of activity on the same set of units. The parallelism provided by the multiple hardware units is used to give a rich microstructure to the individual concepts that correspond to semantic nodes rather than being used to allow many concepts to interact simultaneously. The semantic net formalism can then be seen as a crude description of the interactions between complex patterns of activity. The formalism captures the way in which concepts interact, but it ignores the microstructure of the concepts that is provided by the particular patterns of activity used to represent them in the hardware.

I argue that the second approach is a more promising model of how concepts are represented in the nervous system and that an understanding of the particular patterns of activity used for particular concepts (their microstructure) is important because the interactions between concepts that are formalized as a single link in a



(a)

(JOHN FATHER LEN)
(MARY FATHER LEN)
(JOHN SISTER MARY)
(KATE FATHER JOHN)

(b)

FIG. 6.1. Two formalisms for representing relational information. In (a) the roles of the constituents of a relationship are determined by their positions at the head, tail, or side of an arrow. In (b) the position in a string determines the role.

semantic net are actually generated by millions of simultaneous interactions at the level of their microstructures. An understanding of these microcomputations is the key to understanding how existing concepts are recalled appropriately, how relationships between concepts change, and how new concepts arise.

It is important to realise that the behavior of any network of hardware units can be described either at the level of activities in individual units or at a higher level where particular distributed patterns of activity are given particular names. This is illustrated in Fig. 6.2. Figure 6.2(a) depicts a set of units and the physical interconnections between them. This set is intended to be merely a part of a larger parallel machine. For simplicity each unit is assumed to have two possible states of activity, and Fig. 6.2(b) shows a hypothetical sequence of states of activity of all the units. Figure 6.2(c) shows the sequential relationships between the patterns of activity of the units in Fig. 6.2a. Notice that although the diagramatic formalism is the same in Fig. 6.2(a) and 6.2(c) the interpretation of that formalism is quite different. In Fig. 6.2(a) the nodes correspond to different parts of the machine; links correspond to particular physical connections; and many different nodes can be active at once. None of these descriptions applies to Fig.

6.2(c). Here the different nodes are implemented in the machine as mutually exclusive, distributed representations, that is, different patterns of activity on the same set of units.

We have seen that the very same physical events can be described at two quite different levels. It is an open question as to which level of description is being used when a psychologist uses the semantic net formalism to describe a person's knowledge.

6.2. A PROGRAMMABLE PARALLEL COMPUTER

Artificial intelligence differs from experimental psychology or neurophysiology in its criterion for what counts as a good model. Instead of being primarily concerned with explaining behavioral or neurophysiological data, it aims to produce working programs that can demonstrably perform a task. In domains like

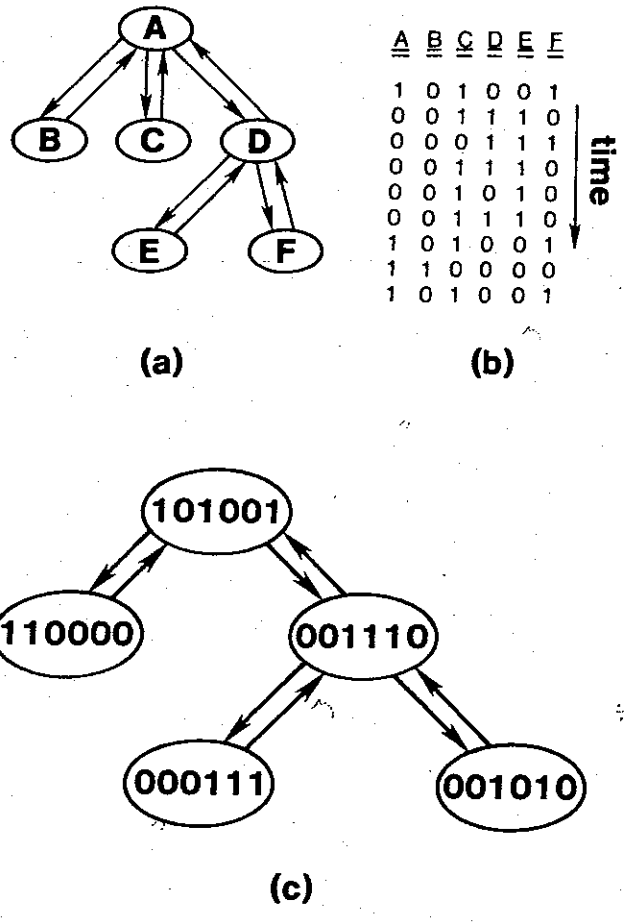


FIG. 6.2. (a) The physical interconnections between some units that form part of a parallel machine. (b) A hypothetical sequence of states of activity of the units in (a) as a result of mutual interactions and input from the rest of the machine. (c) A graphical representation of the transitions between states shown in (b). Although this graph looks like the graph in (a), it has a quite different interpretation.

language understanding or visual perception, this approach has led to many insights about nonobvious difficulties that any information-processing system must overcome. Because of the prevailing technology, however, the goal of generating a working model has typically meant generating working models on existing digital computers. These computers are very different from the brain, and the differences affect the models. This problem can be circumvented by using a digital computer to simulate a different kind of computer that is much more like the brain. The methodology of building working programs can then be preserved, but now the programs are to be designed for the brainlike, simulated computer.

The simulated computer must be similar enough to the brain so that it is reasonable to expect that the computational techniques that work well on it will also work well in the brain. It is probably not necessary, however, to mimic all the details of the neural hardware exactly in order to discover higher-level, software principles that are applicable to a wide range of highly parallel computers composed of many simple computing elements with rich hardware connectivity.

In order to investigate a particular idea about how the semantic net formalism might be implemented in neural nets, a simple but powerful parallel computer was defined and simulated. The computer consisted of a number of perceptron-like units (see Chapter 1, Section 1.2.2). Each unit had an input line from every unit (including itself). At any moment each input line was either active or inactive (1 or 0). Each line had an associated real-valued weight, and a unit produced a 1 as its output at the next moment of time if the sum of the weights on its active input lines exceeded its threshold. Otherwise it produced a 0. The output of a unit determined the states of activity at the next moment of all the input lines emanating from that unit. This type of computer is a particular example of a matrix model (see Chapter 1, Section 1.2.3). The current states of all the units define a binary state vector. The next state vector is generated by multiplying the current one by the matrix of weights and comparing each component of this vector with the corresponding component in the vector of thresholds, as shown in Fig. 6.3(a).

External input to the system was achieved either by setting the initial states of the units or by giving each unit an external input that could have any real value. This value was added to the sum of the weights on the active lines before thresholding.

In programming a computer of this type to perform a given task, several different kinds of decision must be made. First, the total set of units must be conceptually divided into subsets for representing different entities in the task domain. If local representations are used, there will be a single unit for each entity. If distributed representations are used, different states of activity of the same subset of units will represent different entities, and it will be necessary to choose a particular pattern of activity to represent each entity.

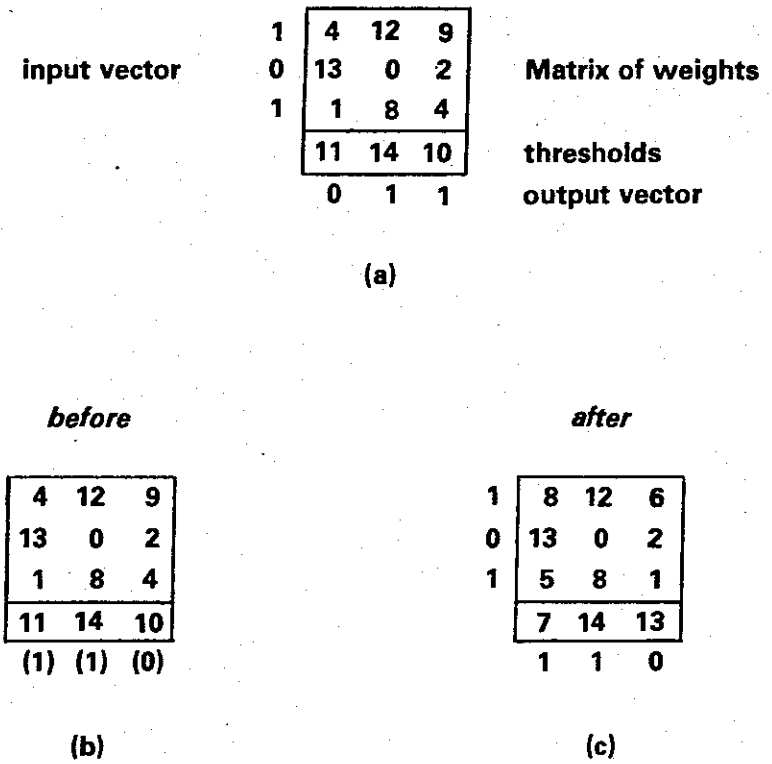


FIG. 6.3. (a) The matrix of weights for a system containing three units. Each column corresponds to a unit. The output of a unit is generated by adding up all the weights in its column that are in rows that have a 1 as input. This sum is then compared with the threshold. (b) A system that is required to give the output shown in parentheses when it gets the input vector shown. It does not do so, so the weights and thresholds must be changed. (c) The set of weights and thresholds that would be obtained by applying the perceptron convergence procedure if the correct outputs are required to be achieved by a margin of at least 6. The first unit in (b), for example, has active weights of 4 and 1 and a threshold of 11, so it fails to "fire" by 6. It is therefore failing to achieve the required margin by 12. So each of the three relevant quantities is changed by 4 in the appropriate direction.

Once these decisions about representations have been made, it is necessary to select the weights in the matrix so that they cause the appropriate sequences of representations (i.e., binary state vectors). This sounds like a tedious and tricky task, but it can be done automatically if the required sequences of binary state vectors are known. The requirement that one state vector should be succeeded by another is equivalent to a set of separate requirements on each unit; namely, that the state of the unit specified by the second state vector should be caused by the inputs to the unit specified by the first state vector. Thus, requiring a set of sequences of state vectors amounts to requiring each individual unit to respond to some patterns of activity of its input lines with a 1 and to other patterns with a 0. The perceptron convergence theorem (see Chapter 1, Section 1.2.2) specifies a way of finding a set of weights that will achieve any required set of responses

to input patterns, provided any such set of weights exists. Figure 6.3 gives a simple example of how the matrix of weights would be changed to make one binary state vector succeed another.

Although the individual units are perceptrons and the perceptron convergence procedure is being used to set the weights, the whole approach is very different from the way perceptrons were originally used. The units "look" at the internal state of the machine not at a perceptual input, and the convergence procedure is used not for learning but for achieving sequences of internal states that have been specified by the programmer.

6.3. FROM SEMANTIC NETS TO STATE VECTORS

The information in any network consisting of nodes connected by labeled, directed arcs is equivalent to a set of triples, each of which consists of two nodes and an arc label. Any device that can produce the missing component of a triple when given the other two elements can be said to contain the information in the semantic network. Cases in which the third component of a triple is not uniquely determined by the other two are particularly interesting and are considered later.

The method that was used for implementing a semantic net in the associative computer defined earlier involved dividing the hardware units into four groups (called assemblies). The first three assemblies were called ROLE1, REL (short for relation), and ROLE2. The states of activity of these assemblies were used to represent the identities of the two nodes and the arc label involved in a relationship. The associative computer could be queried about a particular triple by putting it into an initial state in which two of the first three assemblies had patterns of activity representing two components of the triple, and the remaining assemblies started off with all their units inactive. The computer would complete the triple by settling down into a state in which the missing component of the triple was represented by the state of activity of the relevant assembly.

The fourth assembly was called PROP (short for proposition). For each particular triple stored by the associative computer there was a corresponding particular state of the proposition assembly. Recall of triples from two of their components was achieved by making these states of the PROP assembly have the following properties:

1. Any two of the components of a triple would cause states of the PROP assembly similar to the state that represented the complete triple.
2. States of the PROP assembly that were similar to a state representing a complete triple would cause subsequent states even more similar to the state representing that triple.
3. A state of the PROP assembly representing a particular triple would cause the appropriate states of the ROLE1, REL, and ROLE2 assemblies.

How these three properties of the PROP assembly were achieved is described in the next section after the behavior of a system having these properties has been demonstrated.

Figure 6.4 shows the output of a two-part computer program consisting of a simulator, which simulates a parallel computer, and a handler, which handles the interactions between the user and the parallel computer. The handler translates the first instruction into a set of operations (explained later) that modify the weights of the parallel computer so as to store the three triples (JOHN FATHER LEN), (JOHN SISTER MARY), and (KATE FATHER BILL). The second instruction, (RECALL '(JOHN FATHER O)), tells the handler to set up a particular initial binary state vector in the parallel computer and to print out a description of each subsequent binary state vector. The description is generated by comparing the state of each assembly with a stored list of binary states that have been assigned particular names. If there is no perfect match, the name of the nearest match is used with a numerical suffix indicating the number of places where the match failed. The particular PROP state that represents a complete

```
*(STOREALL '( (JOHN FATHER LEN)
              (JOHN SISTER MARY)
              (KATE FATHER BILL) ))

*(RECALL '(JOHN FATHER O))

  ROLE1      REL        ROLE2        PROP

  JOHN       FATHER     0            0
  JOHN       FATHER     0            JOHNFATHERLEN4
  JOHN       FATHER     LEN          JOHNFATHERLEN
  JOHN       FATHER     LEN          JOHNFATHERLEN

*(RECALL '(JOHN O LEN))

  ROLE1      REL        ROLE2        PROP

  JOHN       0          LEN          0
  JOHN       0          LEN          ?
  JOHN       FATHER     LEN          JOHNFATHERLEN2
  JOHN       FATHER     LEN          JOHNFATHERLEN
```

FIG. 6.4. The output of a program that simulates a parallel computer. The STOREALL instruction causes modifications to the interactions between the hardware units. Each RECALL instruction sets up an initial binary state vector for the parallel computer, which then settles down into a stable state that includes a representation of the initially missing component of the triple. In order to make the input and output intelligible, a separate program translates binary state vectors into words and vice versa. Numerical suffixes on words indicate imperfect matches between the actual binary state vectors and ones with known names (see text).

triple is printed out by concatenating the names for the constituents of the triple. This is just a convention for naming the binary state vector and has nothing to do with the way the simulated parallel computer works. Similarly, the stored lists of named binary states are not part of the simulated computer and are not available to it.

Figure 6.4 shows that an initial state vector representing two components of a triple causes a subsequent PROP state close to the one for that triple, which subsequently settles into exactly the PROP state for that triple. Notice that, in this example, even an imperfect PROP state is sufficient to cause the missing component of a triple.

6.4. CONTEXT-SENSITIVE ASSOCIATIONS AND HIGHER-LEVEL UNITS

In order to store triples effectively in an associative computer, it is necessary to make the relational term of the triple act as a context to which the association between the other two terms is sensitive. Suppose, for example, that we wish to store the four triples shown in Fig. 6.5. It is clear that the first term in the triple does not determine the third term. It all depends on what the relational term is. It is also clear that the relational term by itself does not determine the third term either.

It is fairly easy to show that, in a system of the kind presented here, such context-sensitive associations cannot be achieved if the direct effects of two patterns combine to produce a third pattern. In the example in Fig. 6.5, the required third term is either "one" or "two." The patterns for these two terms must differ somewhere, so let us focus on one unit that is, say, on when the third assembly has the pattern for "one" and off when it has the pattern for "two." Consider how the patterns in the other two assemblies influence this unit. The effect of a pattern in one assembly on a unit in another assembly is simply the sum of the weights on the lines coming from the active units in the pattern. So the patterns for "one" or "two" in the first assembly and the patterns for "same" or "different" in the second assembly will contribute fixed amounts, Q_1, Q_2, Q_s, and Q_d, respectively. For the unit to behave correctly it is necessary that:

(ONE	SAME	ONE)
(ONE	DIFFERENT	TWO)
(TWO	DIFFERENT	ONE)
(TWO	SAME	TWO)

FIG. 6.5. Given this set of four triples, any two components of a triple uniquely determine the remaining component, but simply adding together the effects of the two known components cannot correctly determine the missing one (see text).

$$Q_1 + Q_s > \theta \tag{6-1}$$

$$Q_1 + Q_d \leq \theta \tag{6-2}$$

$$Q_2 + Q_s \leq \theta \tag{6-3}$$

$$Q_2 + Q_d > \theta \tag{6-4}$$

Combining (6-1) with (6-4) yields:

$$Q_1 + Q_2 + Q_s + Q_d > 2\theta \tag{6-5}$$

but combining (6-2) with (6-3) yields:

$$Q_1 + Q_2 + Q_s + Q_d \leq 2\theta \tag{6-6}$$

Hence there is no way of choosing the weights to achieve the desired interactions. This is essentially the same as the proof that a perceptron cannot compute an exclusive-or (Minsky & Papert, 1969).

The importance of this proof is that it shows that simply combining the effects of two patterns does not provide context sensitivity. The only way to achieve this property is by having extra units, which are not used to represent the constituents of a triple and which respond to conjunctions of active units in at least two assemblies. For example, an extra unit might respond to the conjunction of the patterns for "one" and "same" in the first two assemblies. This unit could then cause the pattern for "one" in the third assembly.

The PROP assembly consists entirely of these extra units. The set of patterns to which a unit responds positively can be called its "receptive field." The number of possible receptive fields is extremely large, even for units with a fairly small number of inputs; thus it is out of the question to have all possible receptive fields represented by the units in PROP. Ideally the receptive fields of the extra units should be chosen so as to make the units as helpful as possible in causing the required completions of triples. We return to this issue of selecting helpful receptive fields in Section 6.8. For the simulation, the receptive fields were chosen randomly by assigning random weights to the input lines of the units in PROP.

When a particular triple was coded by the patterns in the ROLE1, REL, and ROLE2 assemblies, it caused a particular pattern of activity in PROP. The mapping from triples to patterns in PROP was random, and the process of storing a particular set of triples in the associative computer did not change this mapping. What did change was the inverse mapping from states of the PROP assembly to states of the other three assemblies. Using the perceptron convergence procedure, the weights that determined this inverse mapping were modified until the three assembly states representing the constituents of each triple were caused by the particular PROP state that those three states gave rise to. This made each triple correspond to a "resonant" state of the whole system because the states of

the first three assemblies caused a state of the remaining assembly, which, in turn, caused them.

The resonance was further increased by using the perceptron convergence procedure to modify the weights between units within the PROP assembly until each PROP state that corresponded to a stored triple caused itself as its own successor (in the absence of any input from other assemblies). The result of associating each PROP state with itself in this way was that states similar to one of the autoassociated PROP states tended to become even more like that state. The reason for this effect is that the state of each unit is jointly determined by the effects of all the other active units, so that if a few units have their states changed, the unchanged active units will tend to change them back again.

The way in which states of the PROP assembly are used in storing triples has similarities to a standard computational technique called *hash coding*. To associate social security numbers with names, for example, one first applies a hashing function to the string of characters that make up the name. The function returns a number that is then used as the address for the memory location in which the associated social security number is to be stored. The way in which states of the first three assemblies are randomly mapped onto states of the PROP assembly is reminiscent of hashing, but there is one major difference. Small changes in the character string typically cause the hashing function to return a totally different address; thus hash coding does not work when the key (the character string) is imperfect, unless, of course, all possible imperfections are considered ahead of time, and the information is separately associated with every variation of the key.

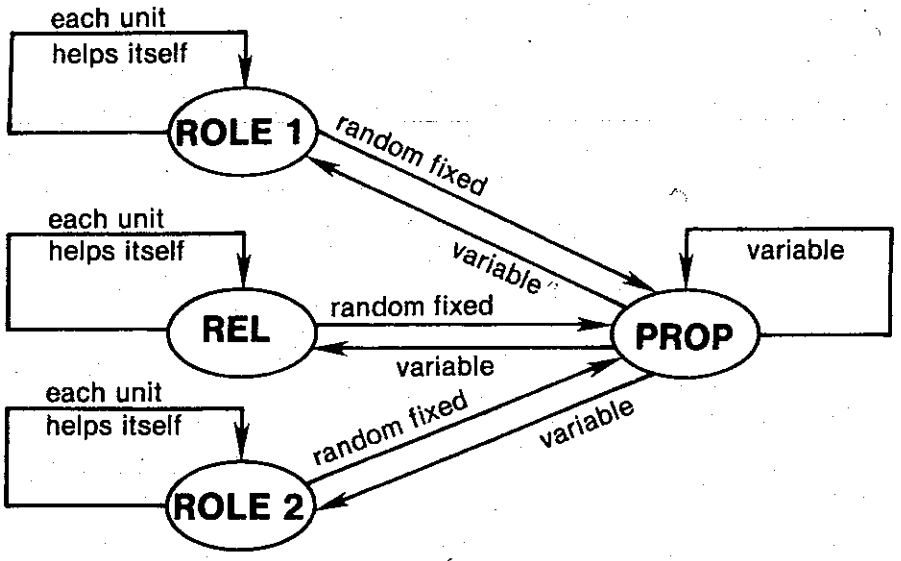
The mapping from triples to PROP states, on the other hand, has the property that similar combinations of states for the first three assemblies lead to similar PROP states. The reason for this is that some of the units in PROP will be driven either far above or far below threshold by the states of the first three assemblies. Therefore although small changes in these assemblies will slightly alter the input to these nonmarginal PROP units, the alterations will be insufficient to change their states. When only two of the constituents of a triple are represented in the first three assemblies, the ensuing PROP state will be somewhat different from the state caused by the complete triple, but it will be more similar to it than to the PROP states corresponding to other triples. It will, therefore, tend to be "captured" by the autoassociated PROP state for the whole triple.

We have now seen how the three properties of the PROP states that were described in section 6.3 were achieved. Figure 6.6 shows the various different ways in which the simulation treated the weights determining the effects of one assembly state on another. The units in the first three assemblies have high thresholds but are also self-excitatory. This causes them to act like flip-flops, so the initial states of activity of these assemblies tend to remain stable during the settling process.

INPUT VECTOR	ROLE 1	REL	ROLE 2	PROP
ROLE 1	positive on diagonal	none	none	random and fixed
REL	none	positive on diagonal	none	random and fixed
ROLE 2	none	none	positive on diagonal	random and fixed
PROP	variable	variable	variable	variable

OUTPUT VECTOR

(a)



(b)

FIG. 6.6. (a) The matrix of weights arranged for the system that stores triples. Many of the submatrices are null. For the three assemblies that code the constituents of a triple, the submatrices determining the effect of an assembly state on itself have all 0 weights except for the leading diagonal. This makes these assemblies retain whatever patterns they are given. The thresholds (not shown) are all positive. (b) An alternative representation of the way in which the weights are initialized and altered.

6.5. PROPERTY INHERITANCE

A major issue in artificial intelligence is how to relate the representations of a token and a type so as to ensure that facts or properties associated with the type are inherited by the token. One method is to make a new copy of the type each time a new token is required. This straightforward approach is space-consuming because all the properties of the type are duplicated for every token. Also if any new knowledge is acquired about the type, it must be explicitly added to all the tokens that already exist. The obvious alternative to making a full copy of the type is to give each token a pointer back to its type, so that whenever a question arises about the token, the type can be inspected. This approach is currently the generally accepted one. It allows property inheritance in hierarchies in which type/token relationships hold between adjacent levels. Suppose, for example, that Clyde is a particular elephant and that the system needs to know how many legs he has. If there is no information about this attached directly to the Clyde representation, then the representation of Clyde's superordinate type, elephant, is checked. If the answer is not attached to the elephant representation, then its superordinate type, quadruped, is examined, and so on.

Property inheritance appears to be a basic characteristic of human conceptual representations, and consequently the ease with which it is achieved by a representational scheme is a measure of the adequacy of the scheme as a model of human knowledge structures. An approach such as Fahlman's (Chapter 5, this volume), which uses local representations in a parallel machine, needs to have specific hard-wired connections between the representations of types and tokens in order to allow property inheritance. If new connections can be created as required (See Feldman, Chapter 2, this volume), Fahlman's scheme is a neat solution to property inheritance. However there is also a solution which does not require specific connections. If distributed representations are chosen appropriately, property inheritance just drops out.

Suppose that the pattern of microfeatures (individual active units) that represents a type is simply the set of microfeatures common to the patterns that represent the tokens of that type. Any effects that are caused by the pattern for the type will automatically transfer to the patterns for the tokens, unless they are overriden by the effects of the additional microfeatures in a particular token. Thus property inheritance is automatic and there is also a way of preventing it in exceptional cases. Furthermore, generalization will be a basic system property, because type patterns will automatically acquire any effects possessed by all their token patterns.

Automatic generalization is not always desirable. For example, the property of being an individual elephant should not be allowed to transfer to the representation of the type "elephant" even though it fits all the tokens. The pattern for the type must be more complex than just the intersection of the microfeatures in

the patterns for the tokens. However, the naive idea that abstraction is just the ommission of the microfeatures that vary within the class seems to have some very useful computational consequences in an associative computer of the type presented here.

Figure 6.7 shows the input and output of the computer program when it is instructed to store three triples and then to complete two of them. Figure 6.8 shows what happens if the program is then asked to complete triples that have not been explicitly stored. Remarkably, it gets the right answers. Figure 6.9 shows why. The pattern for CLYDE contains the pattern for ELEPHANT, so the effects of this pattern are inherited by CLYDE. Similarly, the pattern for BILL contains the pattern for PERSON. Figure 6.10 shows what happens if four triples are stored, including one in which a token, SCOTT, behaves differently from its type, PERSON. The two recall tests show that the exception is learnt and that this does not prevent BILL from inheriting his color from PERSON, though the inheritance becomes slightly shaky.

In a sense, distributed representations finesse the property inheritance problem by making the representation of the type be a constituent of the representation of each token. This solution would lead to wasteful, multiple representations of the type in any system that stored individual representations separately, but this problem does not arise in systems where the ability to recreate an active representation is not achieved by storing that representation or anything like it.

```
*(STOREALL '((ERNIE COLOR GREY)
             (ELEPH COLOR GREY)
             (PERSON COLOR PINK)))

*(RECALL '(ELEPH COLOR 0))

  ROLE 1      REL        ROLE 2      PROP

  ELEPH       COLOR        0           0
  ELEPH       COLOR        0           7
  ELEPH       COLOR      GREY        ELEPHCOLORGREY4
  ELEPH       COLOR      GREY        ELEPHCOLORGREY1

*(RECALL '(PERSON COLOR 0))

  ROLE 1      REL        ROLE 2      PROP

  PERSON      COLOR        0           0
  PERSON      COLOR        0         PERSONCOLORPINK4
  PERSON      COLOR      PINK        PERSONCOLORPINK1
  PERSON      COLOR      PINK        PERSONCOLORPINK
```

FIG. 6.7. Three triples are stored and recall is tested for two of them.

*(RECALL '(CLYDE COLOR 0))

ROLE 1	REL	ROLE 2	PROP
CLYDE	COLOR	0	0
CLYDE	COLOR	0	?
CLYDE	COLOR	GREY	?
CLYDE	COLOR	GREY	ELEPHCOLORGREY2

*(RECALL '(BILL COLOR 0))

ROLE1	REL	ROLE 2	PROP
BILL	COLOR	0	0
BILL	COLOR	0	?
BILL	COLOR	PINK	PERSONCOLORPINK2
BILL	COLOR	PINK	PERSONCOLORPINK

FIG. 6.8. After storing the colors of elephants and people, the system is asked about the colors of CLYDE and BILL. It completes the triples correctly, even though these facts have not been explicitly stored. Figure 6.9 shows why.

*(SHOWSTATES 'ROLE 1)

000000	000000	0
111000	000000	ELEPH
000111	000000	PERSON
111000	111000	CLYDE
111000	000111	ERNIE
000111	101010	SCOTT
000111	010101	BILL

FIG. 6.9. The patterns of activity used to represent various objects in the assembly ROLE 1. The states of the first six units code the type of object (elephant or person). The remaining six units are used to code a particular token by further specifying a type. The similarity between the patterns for a type and a token cause them to have similar effects on the PROP assembly. This causes appropriate generalization.

6.6. TWO TYPES OF CONCEPTUAL STRUCTURE

In order to store a semantic net in the manner described above, it is necessary to choose a particular pattern of activity (i.e., a set of microfeatures) to represent each node in the semantic net. A simple first approach is to choose a random pattern for each node and to ensure that no two nodes are too similar. This allows associations between patterns to be implemented, but it fails to capture the similarities of nodes to one another. For example, the method of achieving property inheritance demonstrated in the previous section relies on a particular

```
*(STOREALL '((ERNIE COLOR GREY)
             (ELEPH COLOR GREY)
             (PERSON COLOR PINK)
             (SCOTT COLOR WHITE)))

*(RECALL '(SCOTT COLOR 0))

  ROLE 1      REL        ROLE 2      PROP

  SCOTT       COLOR        0           0
  SCOTT       COLOR        0           ?
  SCOTT       COLOR      WHITE       SCOTTCOLORWHITE1
  SCOTT       COLOR      WHITE       SCOTTCOLORWHITE

*(RECALL '(BILL COLOR 0))

  ROLE 1      REL        ROLE 2      PROP

  BILL        COLOR        0           0
  BILL        COLOR        0           ?
  BILL        COLOR      PINK          ?
  BILL1       COLOR        ?         PERSONCOLORPINK4

*(SETTLEMORE)

  BILL1       COLOR        ?         PERSONCOLORPINK2
  BILL1       COLOR      PINK        PERSONCOLORPINK3
  BILL1       COLOR      PINK        PERSONCOLORPINK3
```

FIG. 6.10. An exception to the rule that people are pink is included in the list of triples to be stored. This does not prevent the system from correctly answering a query about the color of BILL. However the interference caused by the exception makes the system take longer to settle, and it slightly degrades some of the patterns in the final state.

kind of similarity in which the set of microfeatures for a token contains, as a subset, the microfeatures for the type. This example makes it clear that the ''direct content'' of a concept (its set of microfeatures) interacts in interesting ways with its ''associative content'' (its links to other concepts). The reason for this interaction, of course, is that the associative content is *caused* by the direct content. Each association is implemented by the process of completing a triple, and this process involves creating a resonance which depends on the direct contents of the concepts being associated.

The distinction between direct content and associative content may be an important contribution of this approach to psychology. There is good evidence (Mandler, 1980) that human memory for items involves two separate components, which Mandler has called ''integrative'' and ''elaborative'' structure. The essential idea is that an item has internal coherence as well as external relations to

other items. Within the traditional semantic net framework, it is hard to model this distinction. If, however, integrative structure is interpreted as direct content, and elaborative structure as associative content, it is easy to see that the two are qualitatively different.

The semantic net formalism captures an aspect of conceptual structure (the associations between concepts) that subjects are able to make verbally explicit. It also captures the higher levels of our perceptual representations, which seem to involve structural descriptions in which scenes or objects are decomposed into their constituents (Hinton, 1979; Palmer, 1977; Reed, 1974). These structural descriptions are just semantic nets in which the relationships are spatial. A quite different formalism, in which a concept is only a large set of features, is important in explaining similarity judgments (Tversky, 1977). It is also common in models of pattern recognition, which typically assume that a shape can be represented as a set of features. A battle has raged between these two formalisms. Both sides seem to have assumed that the two formalisms are competing to explain the same phenomena.

It is clear that both formalisms are applicable to the computer simulation presented above. Concepts are patterns of activity (sets of microfeatures), and associations involve interactions between these patterns. The semantic net formalism is a high-level description of the associative content and the feature-set formalism is a low-level description of the direct content. There is no conflict.

Neither formalism alone can capture the way in which direct content influences associative content and vice versa. The property inheritance example shows that the interaction between these two types of content can provide a neat solution to a major problem for semantic nets. The next section shows how another major problem may have a similarly neat solution.

6.7. MEMORY SEARCH AS A CONSTRUCTIVE PROCESS

It is a commonplace in Artificial Intelligence that data structures alone are not enough. There must also be effective search procedures which can find pieces of data that satisfy descriptions (Norman & Bobrow, 1979). A particular node in a semantic net, for example, might be described by saying that it is in relation R to node A and relation S to node B. If many different nodes satisfy each part of the description separately, but only one node satisfies the whole description, then it is nontrivial to find this node.

An obvious but inefficient method is to store separate lists of the nodes satisfying each part of the description and then to find the intersection of the relevant lists. This method requires a lot of extra storage, for the lists, and also involves computing intersections every time a memory search is performed. An

obvious use for parallel hardware is to facilitate memory search. How can this be done in an associative computer?

Scott Fahlman (Chapter 5, this volume) shows how the search problem can be solved efficiently in a parallel computer in which there is a hardware processor for each node in the semantic net. Each part of a complex description is used to place a specific marker on all the nodes which satisfy it. A message is then broadcast to all the nodes telling each of them to report back if it has all the markers. I shall outline a method that is similar in spirit to this marker-passing algorithm but is a natural development of the idea that concepts correspond to patterns of activity rather than to particular hardware units. The method resembles Fahlman's in its use of discrete markers, but it differs in that it dedicates particular pieces of hardware to particular markers rather than to particular semantic nodes. In fact a concept becomes nothing more than a particular bundle of markers (i.e., microfeatures).

Given any semantic net, it would be possible to choose patterns of activity to represent the individual nodes in such a way that every triple was implicitly coded by the presence of specific "relational" microfeatures within the patterns for the relevant nodes. For example, if there was a triple (*A R B*), the pattern for A would include a microfeature that it shared with all the other concepts that stood in relation *R* to *B*. Similarly, the pattern for *R* would have a microfeature that it shared with all other relations holding between *A* and *B*. Each triple would then be coded in two very different ways, once in the interactions between whole assembly states, and once in the direct contents of each of its constituent concepts.

Given this dual encoding, a node satisfying the patterns (? *R B*) and (? *S C*) could be found in three sequential stages. First, an attempt would be made to complete the triple (? *R B*), and this would cause a particular microfeature in the ROLE1 assembly. Next, an attempt would be made to complete (? *S C*), using as the initial state of ROLE1 the pattern caused by the previously attempted completion. The combination of *S* and *C* would cause another particular microfeature in ROLE1, and the search problem would than have been reduced to a pattern completion problem within ROLE1. This subproblem has its own difficulties, but the aim of this section is merely to show how a problem of one type, finding an implicitly specified node in a semantic net, can be reduced to a problem of a different type that an associative computer handles well.

There is one major difficulty with the search scheme as it stands. It requires that each assembly contain a specific unit for each combination of states of the other two assemblies. So although concepts do not require their own units, combinations of pairs of concepts do. This is unacceptable. Fortunately, it is also unnecessary. Instead of using a single microfeature within *A* to code the triple (*A R B*), a small pattern of microfeatures can be used, provided that the microfeatures are carefully chosen.

A semantic network can be said to contain implicit regularities if the nodes and arc labels can be divided into subsets that satisfy constraints. Suppose, for example, that in a particular network, any relational triple that has a concept from the subset S_1 in ROLE1 and a relational concept from the subset S_r in REL, has a concept from the subset S_2 in ROLE2. In this network there is an implicit constraint on the third constituent of any triple whose first two constituents are from the subsets S_1 and S_r, respectively. If the patterns of activity used to represent particular concepts are chosen carefully, this constraint will show up as a simple microinference at the level of the microfeatures. All that is necessary is that for each relevant subset there should be a specific microfeature that is only possessed by the concepts in that subset. If, for example, the members of S_1, S_r, S_2 have the microfeatures f_1, f_r, f_2, respectively, then f_1 in ROLE1 and f_r in REL will imply f_2 in ROLE2. The system can implement this microinference by having a unit in PROP that responds to the conjunction of f_1 and f_r and causes f_2

Notice that f_2 in this example is a more general case of the kind of relational microfeature described above. Instead of coding that a specific other concept is related by a specific relation, f_2 codes the more general constraint that the concept containing it is related to a concept in the set S_1 by a relation in the set S_r. If there are many implicit regularities in the semantic net, there will be many of these more general constraints, and specific conjunctions of them will code specific relations to specific other concepts. This method of encoding specific relationships as conjunctions of more general constraints is much more efficient than using specific relational microfeatures.

The problem of finding a concept that satisfies both (? R B) and (? S C) can now be solved in essentially the same way as before, but instead of causing a single relational microfeature in the ROLE1 assembly, the combination of R and B will cause a number of more general relational microfeatures, corresponding to many different set memberships that can be inferred from the set memberships of R and of B. Similarly, the combination of S and C will cause further microfeatures corresponding to set memberships that can be inferred from those of S and C. If there is a unique answer to the search problem, the two lots of inferred microfeatures should be sufficient to allow microinferences within an assembly to complete the remaining microfeatures of the answer. If there is no unique answer, the inferred microfeatures, and any more that can be inferred from them, will act as a representation of the set of objects that would satisfy the description.

The central idea underlying this search technique is that a particular concept is just a conjunction of set memberships, and that it is possible to infer some set memberships from others. A "search" is performed by using a part of a description to infer some of the set memberships of the desired concept, and then completing the concept by inferring its remaining set memberships. Because the representation of the concept is just this conjunction of set memberships, there is no need for any comparison stage in the search. The idea that remembering involves retrieval of items that are stored as explicit wholes, or that retrieval

descriptions need to be matched against such items is entirely inappropriate to this way of implementing remembering.

Kohonen and his co-workers (Chapter 4, Section 4.4.2, this volume) suggest a rather different method of finding items that satisfy complex descriptions. Each part of the description is used to give subthreshold activation to all the nodes satisfying it, and a threshold is chosen such that only the node that satisfies all parts of the description receives enough activation to exceed the threshold. This is a version of the spreading activation approach (Collins & Loftus, 1975). It is like Fahlman's marker-passing algorithm in that it requires a separate hardware node for each item, but it is not as powerful. Instead of having collections of markers at a node, there is a single number, the activity level, which is a much less informative representation. What we know about the brain makes it seem very probable that human thought involves activation spreading between hardware units, but the whole thrust of this chapter is that the psychological level of description is at a higher level than the hardware, and therefore models that are appropriate at the hardware level may be inappropriate at the psychological level. A single activity level, for example, is incapable of representing an intermediate stage in memory search when some but not all of the microfeatures of a concept have been constructed.

Many of the phenomena that are taken to support a spreading activation model (see Ratcliff, Chapter 10, this volume) are also predicted by the model in which remembering involves constructing the appropriate concept by activating its microfeatures. For example, the constructive model predicts that when errors occur they should be similar concepts, because these share many microfeatures. It also predicts that a concept should prime more general concepts of which it is an instance because the representations of these more general concepts are simply a subset of the microfeatures for their specific instances.

6.8. LEARNING

Within the deliberately restrictive framework of storing and searching sets of triples, there are two rather different learning problems. In order to add a new triple it is necessary to modify the way in which the patterns for the constituents of that triple interact with units in the PROP assembly. The training scheme used by the current computer program is one way of doing this. However, this type of associative learning in which the direct content of the concepts remains unaffected is relatively uninteresting. The major learning problem is to discover how to represent concepts as patterns of activity in such a way that the implicit regularities in the associations between concepts can be captured by microinferences between their microfeatures.

Discovering appropriate microfeatures for a concept is an enormous task because there are so many possible regularities. Given n nodes and r relations,

there are $2^n \cdot 2^r \cdot 2^n$ ways of choosing three subsets between which there might be a simple microinference. For more complex inferences involving more than three microfeatures the number is even greater. If one considers that a really useful microfeature will be one that enters into many different microinferences, the problem becomes even more horrendous.

In conventional implementations of semantic nets learning typically requires extra processes that inspect the representations and notice regularities and exceptions (e.g., Winston, 1975). The approach presented here suggests a very different way of implementing learning. Instead of having a separate system that observes the behavior of the units and adjusts their weights so as to "improve" the microfeatures, each unit can be allowed to alter its own weights so as to become a better microfeature (i.e., one that is involved in more strong microinferences). To do this, each unit must have a measure of how useful its current set of weights makes it as a microfeature. It can then change its weights so as to improve this measure. There is not space here to give a detailed account of how this can be done; and I have not yet applied this kind of self-improvement to the aforementioned program, but I shall give a brief outline of the method in order to dispel the common notion that it is impossible for a local unit to improve its behavior unless it receives a second kind of input that explicitly tells it how well it is doing.

As a concrete example, consider the units in the PROP assembly of the program for storing triples. These units were given random weights on their input lines. This was adequate for a simple demonstration program, but it meant that some of the units in PROP were either always on or always off and were thus useless. The behavior of the system would have been better if the units in PROP had changed their weights so as to be more helpful. What that means, in this context, is that they should have come to respond to common combinations of microfeatures from which microinferences could be made.

Whenever there is an inference of the form: $a \ \& \ b \rightarrow c$, the joint probability of all three events, $p(a \ \& \ b \ \& \ c)$, is higher than the product $p(a \ \& \ b) \times p(c)$, which is the expected joint probability assuming that c is independent of a and b. This means that a unit in PROP will be useful for implementing microinferences if it comes to respond to combinations of microfeatures that occur more often than would be expected from their separate probabilities of occurrence. It is possible to make a unit change its weights so as to latch onto such combinations. All that is necessary is to modify the weights continually so as to "hill-climb" in an appropriate measure of the extent to which the unit is responding to the non-independence among its inputs.

One such measure is the difference between the actual frequency, A, with which the unit is on and the expected frequency, E, with which the unit should be on if its inputs were statistically independent. This measure may be clarified by the following example. Consider a unit with 10 input lines each of which has a probability of 1/2 of being active in any particular trial. Suppose that the first 5

lines are perfectly correlated so that on half the trials they are all on and on the other half they are all off. If the unit develops a threshold of 4.5, and weights of 1 on each of the first five lines, and 0 elsewhere, it will actually "fire" on half the trials. Because, however, it can only fire if all the first 5 lines are active and because each line has a probability of 1/2 of being active, the expected probability of firing, *assuming independence,* is only 1/32. Therefore, A = 1/2, and E = 1/32. There is a large difference between A and E because the particular weights and the threshold cause the unit to respond to a very strong regularity among its inputs.

There are reasons for believing that the optimal measure is neither A − E nor A/E, but this need not concern us here. The main point is that it is possible to estimate A and E and their partial derivatives with respect to the individual weights on the basis of information that is available locally at the unit. This means that there is enough information available locally to hill-climb in a measure that is a function of A and E. Thus by continually making small changes in the weights, the unit can increase the extent to which it responds to regularities among its inputs. I have implemented an algorithm of this kind in several very simple domains and it seems to work well, though the convergence rate is rather slow. This type of weight modification rule is considerably more complicated than the simple correlational synapse suggested by Hebb (1949) and others, but it appears that something of this kind is necessary to allow the direct content of concepts to evolve on the basis of the regularities among their associations.

There are some interesting consequences of making direct content evolve as a result of associative content. If the system has already assimilated a set of concepts and has had enough practice at their associations so that the implicit regularities have become embodied in the microfeatures and microinferences, then a concept that obeys the same implicit regularities will be easy to assimilate. Its associations with existing concepts will allow the existing microinferences to determine a suitable direct content for the concept, and this, in turn, will lead to appropriate generalization.

When an entirely novel set of concepts is introduced, there are serious problems. If the concepts do not contain perceptually specified microfeatures, then there may be a vicious circle. There is no good way to choose the initial direct contents, because the appropriate microfeatures and microinferences cannot evolve until enough associative content has been specified to reveal the implicit regularities. But to enter the associative content, it is necessary to have patterns of activity to represent the concepts. If arbitrary patterns are used to begin with, there will be considerable interference between the various facts. Also, it will be hard to search for concepts satisfying multiple partial descriptions because the search process depends on having appropriate microfeatures.

It is possible to avoid the vicious circle if a new system of concepts is isomorphic to one that has already been thoroughly assimilated. The microfeatures of an existing concept can be used to provide most of the direct content for

the corresponding new one. The existing microinferences will then transfer, so that once some of the correspondences have been specified, the remaining new concepts will receive the appropriate direct content as a result of their associative content and the existing microinferences.

6.9. EXTENSIONS OF THE MODEL

The aim of this chapter is to present a novel way of implementing relational data-structures in parallel hardware. The particular example used is unrealistic in many respects, and the aim of this section is to point out these deficiencies so that the simplifications and ad hoc details in the example are not confused with the principles that it is intended to illustrate. I also indicate the directions in which the model could be extended to cope with some of its major limitations.

6.9.1. Multiple Roles

It is clear that people can represent propositions like "the hippy kissed the debutante in the park," which contain more than three constituents. It would be a trivial extension of the program to allow multiple roles like "location" and "time of occurrence". So long as there is a fixed and relatively small set of discrete roles, it is possible to set aside an assembly for each role. However, this simple approach is inadequate if there are a large number of roles and if some are similar but not identical to others. Consider, for example, the two sentences: "Mary beat John at tennis," and "Mary helped John with his sums." It seems wasteful to have an assembly that is permanently set aside for whatever role the "at tennis" is occupying, and it is unclear whether the sums play the same role in the second sentence as the tennis does in the first. It seems that each relation (verb) has an associated role structure, and that the roles in one structure may be similar to, but not identical with, the roles in another.

It would be possible to extend the program to allow the set of available roles to be determined by the relation involved and to allow two roles in different relations to be similar without being identical. Currently, activity in a unit in a particular role assembly represents that there is an object from a specific set in this particular role. The "receptive field" of the unit therefore covers a number of possible objects but only one role. The extreme specificity of a unit with respect to role is what allows the units to be divided into separate assemblies for the purposes of a higher-level description. This specificity could be relaxed so that activity in a unit represented the three-way conjunction of an object-type, a role-type, and a relation-type. The unit would then respond to any object from a specific set occupying any role from a set of similar roles in the context of any relation from a set of similar relations. It is harder to give a simple higher-level description of the representations in such a system, but it would allow much more

flexibility in the role structures. If two relations had similar roles, the set of units that would be activated by filling a role in one relation with a specific object would be similar to, but not identical with, the set of units activated by filling the similar role in the other relation with the same object.

6.9.2. Procedures and Control Processes

There has been no attempt to model procedural knowledge. All the examples that have been presented of the simulated parallel computer involve a separate conventional program that sets up the initial assembly states and makes the parallel system run for a few iterations. The only thing the parallel hardware does is settle into a stable state, part of which represents the answer to a query. This is a very limited kind of processing. Clearly, a real system would need to have organized sequences of computations. The process of settling into a stable state would then correspond to a single step of a larger computation. In other words, a more complex parallel system would be organized to perform computations by a sequence of settlings. Each such settling would involve a great deal of parallel computation so that the individual settlings in the sequence could implement properties like content-addressable memory. The next section outlines a way of implementing more complex computations that would involve a sequence of settlings at the level of individual proposition representations.

6.9.3. Inference

Memory must allow inference as well as simple retrival of the facts that were explicitly stored. If, for example, we store the facts (JOHN has FATHER LEN) and (LEN has BROTHER BILL), we should be able to complete the triple (JOHN has UNCLE ?). This requires an inference schema of the form (X FATHER Y) & (Y BROTHER Z) $\rightarrow$ (X UNCLE Z). One way of implementing this in parallel hardware is to duplicate the structure used for storing and retrieving propositions but at a higher level. A specific proposition would fill one role in an inference schema in much the same way as an object fills a role in a proposition. There would need to be separate assemblies for simultaneously representing several different proposition/role combinations. However if these assemblies were filled sequentially, it would be unnecessary to duplicate the apparatus for representing individual propositions. The very same apparatus could be used sequentially for retrieving the various individual propositions used in the inference and also for "unpacking" the representation of the inferred proposition into its separate constituent objects. Section 6.10 shows how object roles can be filled sequentially in creating the representation of a proposition, and the same method can be used at a higher level in constructing an inference.

There are, of course, many unsolved problems in implementing inferences in this way. The appropriate inference schema needs to be invoked, and there must

be microinferences at the level of the inference schema that implement the constraints between the bindings of the variables in the constituent propositions. Quantifiers pose problems that have not even been explored in this context. Nevertheless there seems to be no reason to suppose that explicit inference presents an insuperable problem to the kind of memory being proposed.

6.10. LOADING ASSEMBLIES WITH PATTERNS

The program for completing triples requires the known components of a triple to be represented as patterns of activity in the appropriate assemblies. Currently the loading of particular patterns into particular assemblies is performed by the conventional program that handles the parallel simulation. In a more complete parallel system it would be necessary for the loading to be done by the parallel system itself. Suppose, for example, that a query was presented to the system as a sequence of pairs of patterns, one of which represented a component of the triple and the other of which represented the role of that component within the triple. How could the pattern of activity representing the role be used to direct the pattern representing the component to the correct assembly? Kohonen et al. (Chapter 4, Section 4.5.1, this volume) refer to this as the data-switching problem, and they point out that it is hard to solve in the matrix models.

Figure 6.11 shows the output of a program that simulates a parallel machine composed of eight assemblies. The first two are used to represent a component of a triple and its role within the triple. The next three form a filter that is used for solving the data-switching problem. The last three are used to accumulate the serially presented constituents of a triple, and they correspond to the first three assemblies of the previous model. Figure 6.11 shows that the pattern in the OBJECT assembly is copied, after two iterations, into whatever assembly is specified by the contents of the WHROLE assembly. Figure 6.12 shows that if the constituents of a triple are known, the pattern in the WHROLE assembly determines which constituent gets copied into the object assembly.

The program works by using the assemblies TROLE1, TREL, and TROLE2 as a "skeleton" filter. The units in these assemblies have high thresholds, but a particular pattern of activity in WHROLE provides enough excitatory input to all the units in one of the assemblies in the filter so that further excitatory input from other assemblies is enough to turn particular units on. Units in the other two assemblies of the filter do not receive excitatory input from WHROLE. They therefore remain so far below threshold that none of them are turned on by the effects of the patterns of activity in the other assemblies. Each of the assemblies in the filter acts as a selective channel that allows mutual interaction between the object assembly and one of the last three assemblies. The pattern in the WHROLE assembly has the effect of opening up just one of these channels.

The interactions between units are set up so that a pattern in the object assembly will tend to cause a corresponding pattern in each of the filter as-

```
*(ZEROALLASSEMBLIES)
*(INPUTPAIR '(JOHN ROLE1))
*(SETTLE)
```

OBJECT	WHROLE	TROLE 1	TREL	TROLE 2	ROLE 1	REL	ROLE 2
JOHN	ROLE 1	0	0	0	0	0	0
JOHN	0	JOHN	0	0	0	0	0
JOHN	0	0	0	0	JOHN	0	0
JOHN	0	0	0	0	JOHN	0	0

```
*(INPUTPAIR '(MARY ROLE2))
*(SETTLE)
```

OBJECT	WHROLE	TROLE 1	TREL	TROLE 2	ROLE 1	REL	ROLE 2
MARY	ROLE 2	0	0	0	JOHN	0	0
MARY	0	0	0	MARY	JOHN	0	0
MARY	0	0	0	0	JOHN	0	MARY
MARY	0	0	0	0	JOHN	0	MARY

```
*(INPUTPAIR '(MOTHER REL))
*(SETTLE)
```

OBJECT	WHROLE	TROLE 1	TREL	TROLE 2	ROLE 1	REL	ROLE 2
MOTHER	REL	0	0	0	JOHN	0	MARY
MOTHER	0	0	MOTHER	0	JOHN	0	MARY
MOTHER	0	0	0	0	JOHN	MOTHER	MARY
MOTHER	0	0	0	0	JOHN	MOTHER	MARY

FIG. 6.11. The first instruction puts all eight assemblies into the null state. The instructions following fix the initial states of the first two assemblies and cause the system to iterate until it reaches a stable state. States of the OBJECT assembly and the last three assemblies are stable because the units in these assemblies have high thresholds but are strongly self-excitatory. States of the other assemblies are transitory because the units are not self-excitatory.

semblies and vice versa. Also, a pattern in any of the filter assemblies will tend to cause a corresponding pattern in one of the last three assemblies and vice versa. Thus the pattern in WHROLE can direct the flow of information between the first assembly and the last three by selectively opening one of the three channels provided by TROLE1, TREL, and TROLE2.

The mechanism is called a skeleton filter because it works by enabling a skeleton subset of the units in the filter. It is similar in spirit to the more sophisticated filter described by Sejnowski (Chapter 7, this volume).

Figure 6.13 shows that the system can handle sequential input as fast as it can be presented. Although it takes two iterations for input to pass through the filter, successive inputs can be "pipelined" so that a sequence of N inputs only requires $N + 1$ iterations.

```
*(SETINITIAL '(0 0 0 0 0 JOHN MOTHER MARY))
*(INPUTPAIR '(0 ROLE1))
*(SETTLE)
```

OBJECT	WHROLE	TROLE 1	TREL	TROLE 2	ROLE 1	REL	ROLE 2
0	ROLE 1	0	0	0	JOHN	MOTHER	MARY
0	0	JOHN	0	0	JOHN	MOTHER	MARY
JOHN	0	0	0	0	JOHN	MOTHER	MARY
JOHN	0	0	0	0	JOHN	MOTHER	MARY

```
*(INPUTPAIR '(0 REL))
*(SETTLE)
```

OBJECT	WHROLE	TROLE 1	TREL	TROLE 2	ROLE 1	REL	ROLE 2
0	REL	0	0	0	JOHN	MOTHER	MARY
0	0	0	MOTHER	0	JOHN	MOTHER	MARY
MOTHER	0	0	0	0	JOHN	MOTHER	MARY
MOTHER	0	0	0	0	JOHN	MOTHER	MARY

FIG. 6.12. If a triple is present in the last three assemblies, the initial state of the WHROLE assembly determines which component of the triple is transferred to the OBJECT assembly. It does this by temporarily opening one of the three channels provided by TROLE1, TREL, and TROLE2.

```
*(INPUTSEQUENCE '((JOHN ROLE 1) (MARY ROLE 2) (MOTHER REL)))
```

OBJECT	WHROLE	TROLE 1	TREL	TROLE 2	ROLE 1	REL	ROLE 2
JOHN	ROLE 1	0	0	0	0	0	0
MARY	ROLE 2	JOHN	0	0	0	0	0
MOTHER	REL	0	0	MARY	JOHN	0	0
?	0	0	MOTHER	0	JOHN	0	MARY
?	0	0	0	0	JOHN	MOTHER	MARY

FIG. 6.13. The INPUTSEQUENCE instruction causes a particular sequence of pairs of states in the first two assemblies. This sequential input causes the components of a triple to be accumulated in the last three assemblies.

6.11. SUMMARY

In our attempts to understand how information is stored and processed in the brain we are limited by our knowledge of the brain's hardware, by our knowledge of what people can do, and by our ideas about possible ways of organizing information processing in parallel hardware. This chapter has tried to add to the repertory of ideas about information processing by demonstrating a novel way of storing and searching complex, flexible data-structures in highly parallel hardware.

Current digital computers implement flexible data-structures by using pointers. One data-structure is given a link to another by being given its address. This scheme depends on the addressing mechanism and the storage of different data-structures in different places. An alternative scheme which has been proposed by Minsky (1980), Fahlman (Chapter 5, this volume), and Feldman (Chapter 2, this volume) is to replace the addresses by real hardware connections. This removes the need for an addressing mechanism, but it makes it hard to see how new connections can be established rapidly. A third possibility is that concepts are represented by large patterns of activity, and that data-structures are stored by modifying the interactions between these patterns. The aim of this chapter has been to show that this idea is feasible by building a very simple working model.

ACKNOWLEDGMENTS

I would like to thank Eileen Conway, Ed Hutchins, Don Norman, Chris Riesbeck, and Dave Rumelhart for their help. The preparation of this chapter and the research reported in it were performed while I was a Visiting Scholar with the Program in Cognitive Science, at the University of California, San Diego, supported by a grant from the Sloan Foundation.

REFERENCES

Collins, A. M., & Loftus, E. F. A spreading activation theory of semantic processing. *Psychological Review*, 1975, *82*, 407-428.

Hebb, D. O. *Organization of behavior*. New York: Wiley, 1949.

Hinton, G. E. Some demonstrations of the effects of structural descriptions in mental imagery. *Cognitive Science*, 1979, *3*, 231-250.

Levin, J. A. *Proteus: An activation framework for cognitive process models (ISI/WP-2)*. Marina Del Rey, California: Information Sciences Institute, 1976.

Mandler, G. Recognizing: The judgment of previous occurrence. *Psychological Review*, 1980, *87*, 252-271.

Minsky, M. K-lines: A theory of memory. *Cognitive Science*, 1980, *4*, 117-133.

Minsky, M., & Papert, S. *Perceptrons*. Cambridge, Mass.: MIT Press, 1969.

Norman, D. A., & Bobrow D. G. Descriptions: An intermediate stage in memory retrieval. *Cognitive Psychology*, 1979, *11*, 107-123.

Palmer, S. E. Hierarchical structure in perceptual representation. *Cognitive Psychology*, 1977, *9*, 441-474.

Reed, S. K. Structural descriptions and the limitations of visual images. *Memory & Cognition*, 1974, *2*, 329-336.

Tversky, A. Features of similarity. *Psychological Review*, 1977, *84*, 327-352.

Winston, P. H. Learning structural descriptions from examples. In P. H. Winston (Ed.), *The Psychology of Computer Vision*. New York: McGraw-Hill, 1975.