WIS-CS-199-74
COMPUTER SCIENCES DEPARTMENT
The University of Wisconsin
1210 West Dayton Street
Madison, Wisconsin 53706

Received December 27, 1973

A HIERARCHICAL NET-STRUCTURE LEARNING SYSTEM FOR PATTERN DESCRIPTION

by

Harold Addison Williams, Jr.

Computer Sciences Technical Report #199

January 1974

A HIERARCHICAL NET-STRUCTURE LEARNING SYSTEM FOR PATTERN DESCRIPTION

by

Harold Addison Williams, Jr.

ABSTRACT

This thesis discusses a computer program that recognizes and describes two-dimensional patterns and the subpatterns composing those patterns, outputting names, locations and sizes of both patterns and subpatterns. The program also recognizes all patterns in a scene consisting of several patterns.

Patterns are stored in a hierarchical, net-structure permanent memory, which is completely learned as a result of simple feedback from a trainer. Weighted links between memory nodes represent subpattern/pattern relationships. The memory is homogeneous, for subpatterns are represented in terms of primitive features in the same manner that patterns are represented in terms of subpatterns. A short term memory is used to store instances of permanent memory information during recognition.

Pattern recognition is accomplished with a serial heuristic-search algorithm, unusual for a pattern recognition program, which attempts to search memory and compute input properties efficiently. Without special processing, the program can be asked to look for all occurrences of a specified pattern in an input scene.

A HIERARCHICAL NET-STRUCTURE LEARNING SYSTEM FOR PATTERN DESCRIPTION

Harold Addison Williams, Jr.

Under the supervision of Professor Leonard Uhr

This thesis discusses a computer program that recognizes and describes two-dimensional patterns and the subpatterns composing those patterns, outputting names, locations and sizes of both patterns and subpatterns. The program also recognizes all patterns in a scene, for it does not distinguish between recognition of parts of a single pattern and recognition of whole patterns that are part of a scene.

Patterns are stored in a hierarchical, net-structure permanent memory, which is completely learned as a result of simple feedback from a trainer. Weighted links between memory nodes represent subpattern/pattern relationships. The memory is homogeneous, for subpatterns are represented in terms of primitive features in the same manner that patterns are represented in terms of subpatterns. A short term memory is used to store instances of permanent memory information during recognition.

Pattern recognition is accomplished with a serial heuristic-search algorithm, unusual for a pattern recognition program. This algorithm, which attempts to search memory and compute properties of the input in an efficient manner, uses learned information

stored in permanent memory as well as information accumulated in short term memory during the course of recognition. Without special processing, the program can be asked to look for all occurrences of a specified pattern in an input scene.

The program learns by both adding information to and deleting information from the memory net, and adjusting weights associated with information already stored in memory. Learned weights indicate the importance of a part in describing a whole, and the probability of a whole, having found a part. The program tries to discover combinations of memory nodes that will perform well in describing patterns. New memory nodes, stored in an intermediate memory, are generated to represent such combinations. After a testing period, they are either added to permanent memory or discarded. Learning depends on both the trainer-supplied feedback and the information accumulated during recognition.

The program does not assume that any specific primitive feature tests will be used to compute properties of input patterns. Rather, any primitives meeting certain general requirements may be used. The program has been tested using three different types of primitives: letters of the alphabet, matrix templates, and straight line segments. Straight line drawings of simple geometric objects, patterns containing such objects as subpatterns, and scenes containing whole patterns have been recognized.

The program is implemented in LISP 1.5 on a Univac 1108

computer at the University of Wisconsin. In addition to those features already included in the running program, several extensions are described in detail, including mechanisms to handle context and imperfect patterns.

Leonard Uhr

Professor in charge of thesis

ACKNOWLEDGEMENT

I wish to express my deep gratitude to my thesis advisor, Professor Leonard Uhr, for his insightful guidance and encouragement over the past three years.

I am also grateful to my thesis readers, Professors Larry Travis and Raymond Moore for their helpful suggestions.

In addition, I would like to thank Rick LeFaivre and Eric Norman of the Madison Academic Computing Center staff for their conscientious response to my LISP suggestions and questions.

Finally, I am indebted to both my wife Cathy, for her patience and aid in correcting my more unusual grammar, and our cat Arthur, for carefully inspecting the manuscript.

This research was partially supported by NSF grant number GJ-36312 and NIMH grant number MH-12266.

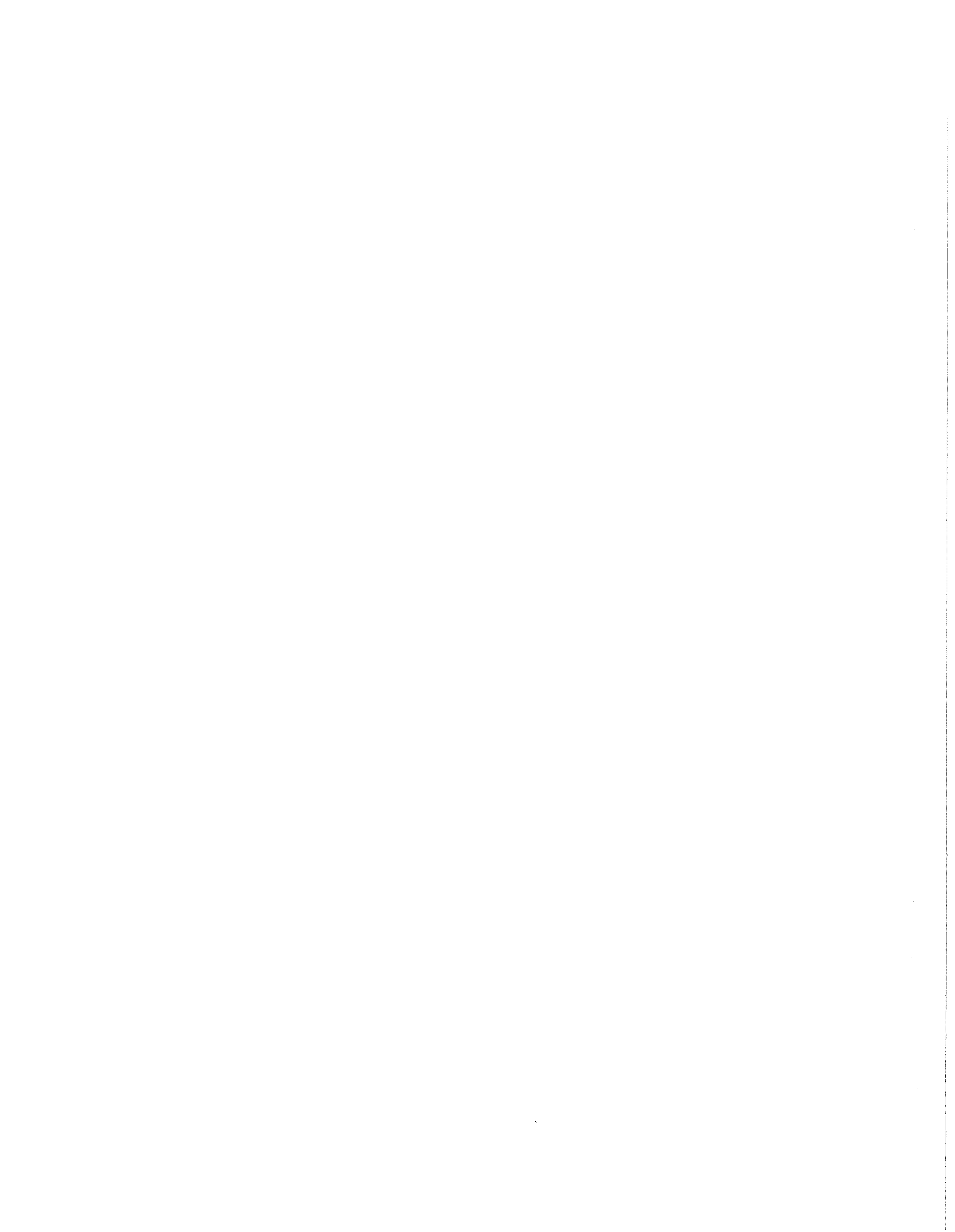


TABLE OF CONTENTS

I. INTRODUCTION

1. An Introduction to the Program HPL 1

    1.1 Examples of Pattern Recognition by HPL . . . 2

    1.2 Primitives 4

    1.3 The Memory Structure of HPL 5

    1.4 The Recognition Algorithm 7

    1.5 Learning 8

    1.6 The Current Status of HPL 9

2. Motivating Factors in the Design of HPL 9

    2.1 Learning 9

    2.2 Memory Structure 10

    2.3 Efficiency 10

    2.4 Human Perception 11

II. REVIEW OF PREVIOUS RESEARCH

1. Classification Techniques of Pattern Recognition . 12

    1.1 The Parallel Classification Technique 12

    1.2 The Sequential Classification Technique . . . 19

2. Descriptive Techniques of Pattern Recognition . . 21

    2.1 Syntactic Techniques of Pattern Recognition . 21

    2.2 Scene Analysis Programs 26

3. Net-Structure Approaches to Pattern Recognition . 28

3.1 The Winston Program 28

3.2 Pattern Recognition Techniques of Uhr 29

3.3 Becker's Model of Intermediate Level Cognition 32

III. A GENERAL DESCRIPTION OF THE PROGRAM HPL

1. An Introduction to HPL 34

2. Information Flow Diagrams of the Processes of HPL 36

3. The Memory Structure 38

4. The Recognition Process 44

5. The Learning Process 47

6. The Context Problem 48

IV. THE MEMORY STRUCTURE

1. The Characterizer 50

1.1 The Description-Set 51

1.2 The Implication-Set 56

1.3 The Relationship between Descriptors and Implicands 56

1.4 Additional Information Associated with a Characterizer Name 57

1.5 Differences in the Representation of Primitives, Compounds and Goals 60

1.6 Some examples of Characterizers 60

2. The Instance 62

2.1 The Description-Set of an Instance 63

2.2 The Implication-Set of an Instance 64

2.3 An Example: Matching Instances to Descriptors and Implicands 65

2.4 Tolerance: Not Requiring an Exact Match . . . 67

2.5 Additional Items Stored in an Instance . . . 68

V. THE RECOGNITION PROCESS

1. Introduction 69

1.1 Definition of Terms 69

2. An Overview of the Recognition Process 71

3. An Example of the Recognition Process 73

4. Weights Associated with an Instance 74

5. Implicand-Processing 77

6. Descriptor-Processing 81

7. Starting the Recognition Process 83

8. Terminating the Recognition Process 85

9. Response and Feedback 85

VI. THE LEARNING PROCESS

1. Introduction 89

2. The Significance of Compounds 89

2.1 Conditional Probabilities and Compounds . . . 90

2.2 Compounds and Allocation of Effort 93

2.3 Compounds May Be Good Characterizers 94

2.4 Compounds Provide Flexibility 95

2.5 Compounds versus Goals 95

3. Inductive Weight Adjustment 96

4. Adding and Deleting Links 98

5. Compounds: Their Formation and Evaluation 99

VII. EXAMPLES OF PATTERN RECOGNITION BY HPL

1. The Choice of Primitives 102

1.1 Letter Primitives 102

1.2 Submatrix Primitives 103

1.3 LINE Primitives 107

1.4 The Relationship between LINE and Submatrix Primitives 109

2. Simple Pattern Recognition 111

3. The Representation of a Pattern 114

4. Recognition of Complex Patterns 116

5. Scene Analysis 119

6. Overlapping Patterns 119

7. Attending to a Pattern: The Needle in the Haystack 123

8. Evaluation of Results 124

VIII. EXTENSIONS OF HPL

1. Introduction 127

2. Expanding the Structure of the Characterizer . . . 127

2.1 Absolute and Relative Attributes 128

2.2 Multi-Valued Functions 131

2.3 Comparison of Characterizers 133

2.4 Weighted Characterizer Arguments 135

3. Partially Found Characterizers 136

3.1 Occluded Patterns 140

4. Relations 140

5. The Context Problem 144

5.1 Guzman's Approach to Context 144

5.2 An Approach to Context within HPL 145

5.3 Learning Context Weights 148

IX. CONCLUSIONS AND FURTHER EXTENSIONS OF HPL

1. Introduction 151

2. A Comparison of HPL with the Work of Winston . . . 151

2.1 Unweighted Links versus Weighted Links
in a Net-Structure 156

2.2 Negative Description Weights 157

3. Further Extensions of HPL 160

3.1 Curved Lines 160

3.2 Two-Dimensional Projections of Three-Dimensional Patterns 161

3.3 General to Specific Recognition 163

3.4 Language Facilities Used in HPL 165

4. Concluding Remarks 167

REFERENCES 170

CHAPTER 1

INTRODUCTION

1. An Introduction to the Program HPL

This thesis discusses an adaptive pattern recognition program called HPL,* that recognizes and describes patterns and subpatterns composing those patterns, outputting names, locations and sizes of both patterns and subpatterns. HPL will also recognize all patterns in a scene, for HPL does not distinguish between recognition of parts of a single pattern and recognition of whole patterns that are part of a scene. Patterns are stored in a unified, hierarchical net-structure memory which is learned as a result of feedback from a trainer. Weighted links between memory nodes represent subpattern/pattern relationships. The recognition process is a serial heuristic search algorithm, which attempts to efficiently search memory and compute features of a pattern. Besides recognizing all patterns and subpatterns, HPL can be asked to look for all occurrences of a *specified* pattern in a scene. HPL is a running computer program. It is also a general framework for discussing further issues in pattern recognition, including context and explicitly-named relations between parts of a pattern.

In this section, we shall briefly describe the components of HPL, including examples of patterns HPL has recognized. The following section discusses several factors that motivated our work.

---

*The acronym means "Hierarchical Pattern Learner".

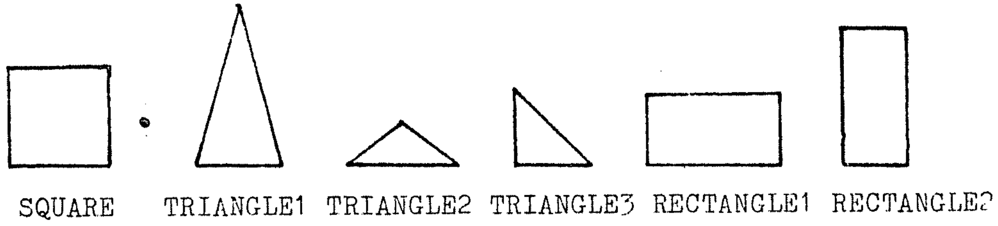
1.1 Examples of Pattern Recognition by HPL

HPL recognizes idealized straight line drawings exhibiting subpattern/pattern (or part/whole) relationships. The input is simplified so that we may concentrate on specific issues, including part/whole recognition, memory representation, context, and the recognition algorithm. In Chapters 8 and 9, we discuss extensions to increase HPL's recognition power. Chapter 7 examines and discusses the examples of pattern recognition shown here, along with additional examples.

Figure 1-1(a) shows the simplest type of pattern HPL recognizes, containing no named subpatterns. The name associated with each pattern is given to HPL as feedback.

HPL can be taught that TRIANGLE1, TRIANGLE2, and TRIANGLE3 are examples of a more general pattern concept called TRIANGLE, shown in Figure 1-1(b). After sufficient training, when given an example of TRIANGLE1, HPL will respond "TRIANGLE1" and then "TRIANGLE".

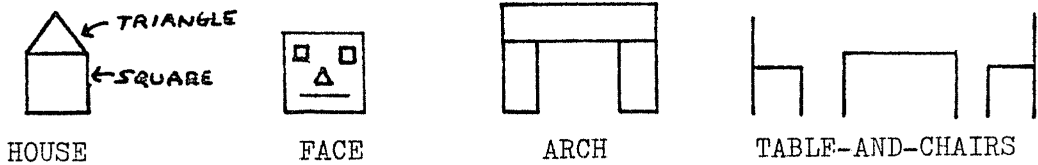
Figure 1-1(c) shows more complex patterns containing the simpler patterns as subpatterns, which HPL can learn and then recognize. There is no limit on the level of hierarchical compounding of patterns. When given the pattern named FACE, HPL will respond with the name of each subpattern it recognizes, its location, and its size, for example, TRIANGLE at location (5,5) of size 0.2, etc., and finally FACE at location (0,0) of size 1.0. Recognized subparts of a scene having no



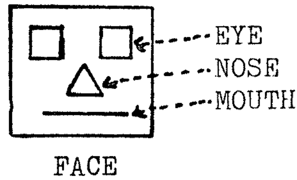
(a) Simple patterns without subpatterns

TRIANGLE = TRIANGLE1, TRIANGLE2, <u>or</u> TRIANGLE3

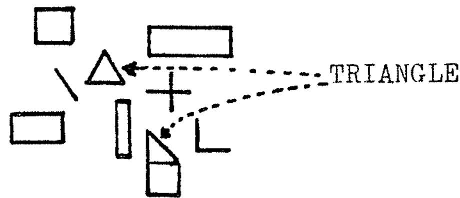
(b) A simple concept



(c) Patterns with simpler patterns as subparts



(d) The context problem



(e) The needle in the haystack problem

Figure 1-1. Some examples of pattern recognition by HPL.

overall name will in a similar manner be output, for example, CHAIR, TABLE and CHAIR, with their respective locations and sizes.

Figure 1-1(d) illustrates the "context problem". Subparts of the pattern FACE, such as EYE or NOSE, might be represented by a square or triangle. We do not want HPL to respond "NOSE" whenever it recognizes a triangle in <u>any</u> pattern, but only when FACE is recognized as well. A modification of HPL to handle this important problem is described in Chapter 8.

Figure 1-1(e) shows the "needle in the haystack" problem. HPL may be instructed to look for any specific pattern, e.g. TRIANGLE, in a scene and will respond with the location and size of each TRIANGLE it finds. No special mechanisms are necessary in order to handle this problem.

1.2 Primitives

<u>Primitives</u> (or primitive features) are built-in (unlearned) functions which extract information from the input, interfacing between the input and the processes of HPL. HPL makes few assumptions about the form primitives must have. A primitive can be any function which computes whether a property is either present or not present in the input at a specified location with a certain size.

Although HPL can be used with any primitive meeting the above requirements, some specific primitives had to be chosen in order to demonstrate recognition. Two different types of primitives were

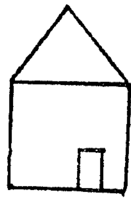
used in most runs of HPL, submatrices and straight lines. (A third type, letters of the alphabet, was used for initial testing.)

Submatrix primitives are M x N matrices consisting of cells of 0 or 1. They are templates that are compared with an M x N region of the input pattern, which is represented as a larger matrix of cells of 0 or 1.

Straight line primitives are built-in straight line recognizers. Because of limitations in computer time, most computer runs were made with this type of primitive; corresponding inputs were well-formed two-dimensional straight line drawings, such as the examples in Section 1.1. These inputs were presented in a special coded format: a list of vertices in the pattern or scene, their locations, and a list of vertices to which they are connected by straight lines.

1.3 The Memory Structure of HPL

HPL's memory is a net-structure in which patterns are represented as nodes and subpattern/pattern relationships are represented as weighted links between nodes. The memory is unified, for there is no structural difference between the way information is stored at the lowest level, e.g., describing a TRIANGLE in terms of the straight lines which compose it, and at higher levels, e.g., describing a HOUSE as a TRIANGLE, RECTANGLE, and SQUARE in certain relative positions. Figure 1-2 shows a HOUSE and its memory representation, in simplified form.



HOUSE

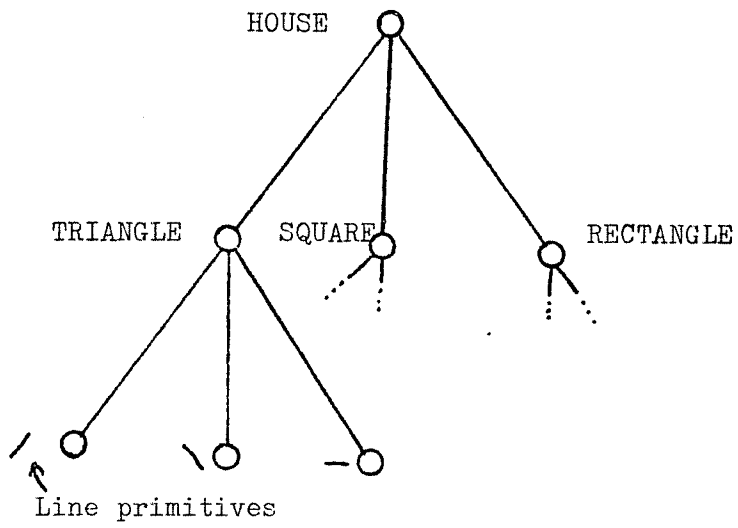


Figure 1-2. A HOUSE and its simplified memory representation.

Two weights are associated with memory links. One weight indicates the importance of a node in describing a node connected to it. For example, the RECTANGLE in the pattern HOUSE might be less important than the SQUARE and TRIANGLE in determining whether HOUSE is present. Another weight indicates the probability of a node, given that a node connected to it has been found. For example, if a SQUARE has been found, a weight gives the probability of HOUSE. This information is used by the recognition algorithm in deciding what to do next.

HPL contains both a permanent (learned) memory and a short term or "active" memory. In permanent memory, locations and sizes are relative to one another. In active memory, location and size are absolute as stored in instances of permanent memory information. This approach saves permanent memory space by not using permanent storage for each possible absolute location and size in the input.

1.4 The Recognition Algorithm

HPL's recognition process is a serial heuristic search algorithm. The use of a heuristic search technique is somewhat unusual for a pattern recognition program, although common in game playing and theorem proving programs. HPL tries to search memory and compute primitives in an efficient and effective order. Its decision as to what step to take next depends on what information has already been discovered in the input and what is stored in memory. For example, if a SQUARE has been found, there will be a

high probability that patterns containing SQUARE, such as HOUSE, will be investigated. If HOUSE is investigated, then subpatterns of HOUSE, e.g., TRIANGLE, will have a high probability of being investigated. The probability of a node, its computation cost and value in contributing to recognition, and the degree of success in attempting to find it, all contribute to the decision-making process. Short term instances of long term memory information are continually updated to reflect the course of the recognition process.

By increasing the value of a node representing a desired pattern, the recognition algorithm is easily biased to look for that pattern in an input, for example, looking for all occurrences of a TRIANGLE in an input (the "needle in the haystack" problem). Thus HPL is influenced in both a top-down manner by patterns to which it is attending and a bottom-up manner by the information found in the input scene.

1.5 Learning

Except for primitives, HPL's memory is entirely learned. Learning consists of forming memory nodes and links between nodes, and adjusting weights associated with links.

The operation of HPL consists of a sequence of "frames", each of which is the following sequence of events. A pattern or scene is presented to HPL as an input. HPL begins the recognition process and outputs the name, location and size of each pattern or subpattern it can identify, or "I DON'T KNOW". After each response,

the trainer may give feedback to HPL, either "YES" or "NO", or the name and location of some pattern in the input, depending on the correctness of HPL's response or the failure to name a pattern that was present. Following the recognition process, HPL enters the learning process, modifying its memory by use of the information accumulated during recognition and feedback.

1.6 The Current Status of HPL

HPL is a 1500 line computer program written in the list-processing language LISP 1.5 [McCarthy, 1962], running on the Univac 1108 computer (which has a 750 nanosecond cycle time) at the University of Wisconsin. The program operates interactively with a human trainer providing feedback or in a batch mode in which the trainer is simulated.

In addition, HPL provides a framework for discussing ideas not implemented in the current program. These ideas concern ways to improve HPL's descriptive ability, e.g., handling general relations and context, and are discussed in Chapters 8 and 9.

2. Motivating Factors in the Design of HPL

2.1 Learning

A primary motivation in designing HPL was that it should be adaptive, that its knowledge about patterns and their structure should be learned rather than built-in. First of all, we are

interested in adaptive mechanisms for their own sake. Secondly, having the program learn pattern descriptions frees its designer from having to decide the nature of such descriptions. Finally, we believe a redundant, weighted, flexible net-structure memory is very appropriate for storing descriptive information about patterns, and such a memory is particularly amenable to learning.

2.2 Memory Structure

We desired a simple, general memory structure capable of expressing information about pattern descriptions in terms of subpattern/pattern relationships. We wanted a unified memory, structurally the same whether simple patterns are described in terms of primitives or complex patterns are described in terms of subpatterns. The memory should be general enough to allow for possible extensions. Finally, the memory should be amenable to learning. These requirements suggested a weighted net-structure memory.

2.3 Efficiency

Adaptive programs may require large amounts of computer time in order to exhibit any interesting behavior. We wanted a program which would be efficient enough to recognize idealized patterns without excessive computing costs.

The issue of efficiency is not irrelevant to artificial intelligence research. If programs could perform unlimited search

without regard to time or memory, many artificial intelligence problems would already be solved. For example, chess is finite and easily handled by an exhaustive search. There is an important difference between the theoretically possible and the practically possible. Although it may be argued that one should not be overly concerned with such issues since present hardware will undoubtedly improve, it is obvious that even current hardware is not fully utilized, and future hardware will never be good enough for exhaustive algorithms.

Efficiency is especially important in pattern recognition, where an input potentially contains a very large amount of information, and there are many possible patterns.

2.4 Human Perception

HPL is not a simulation of human perception; however, attributes of human perception provide good working goals for a mechanical recognizer. Our work has been influenced by our intuitions about human perception, including the following: Human memory is a hierarchical net-structure which contains descriptive information about patterns and is largely learned. Attention and context are important aspects of perception.

CHAPTER 2

REVIEW OF PREVIOUS RESEARCH

1. Classification Techniques of Pattern Recognition

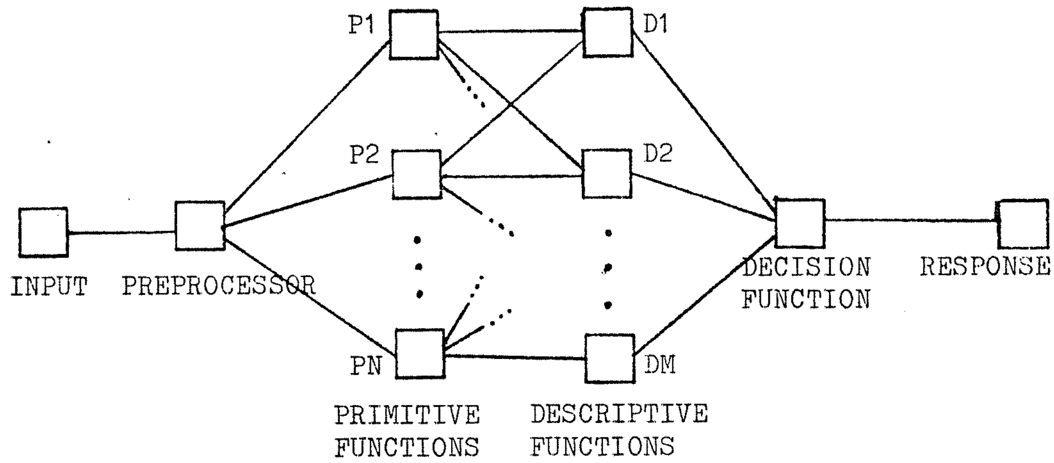
Pattern classification programs attempt to assign a single unique name to an input. They have been either parallel or serial with respect to the computation of primitives, the lowest level "tests" on the input. The most common pattern recognition paradigm is the parallel classification technique.

1.1 The Parallel Classification Technique

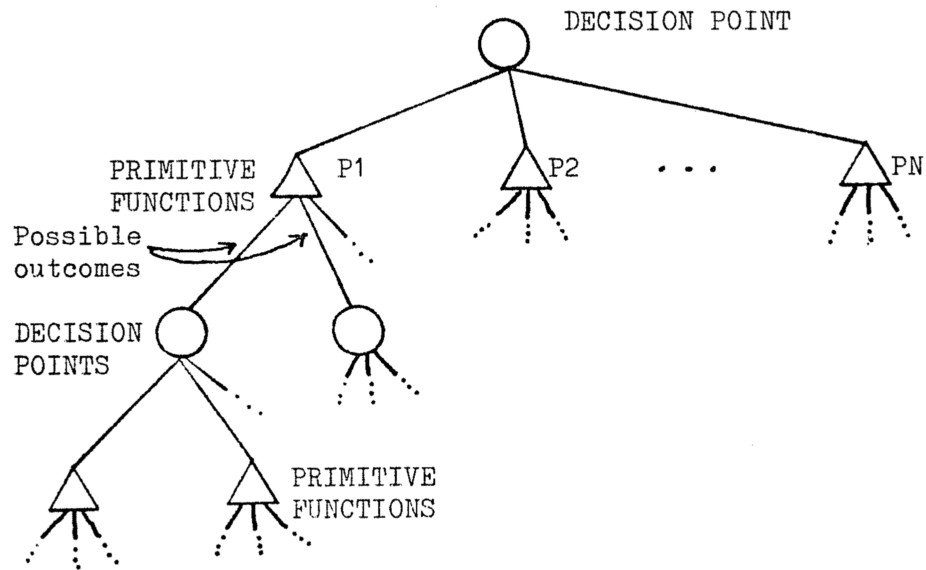
The parallel classification technique is illustrated in Figure 2-1(a). The input pattern may first be preprocessed to eliminate noise, smooth edges, enhance contours and the like. Then a set of primitive functions or tests is applied to the pattern. The values of these functions are usually real numbers indicating the presence or absence of some feature. There is usually one "descriptive function" associated with each possible pattern name. The most common form of this function is a linear evaluation function

$$d_j = \sum c_{ij} \cdot a_i$$

where each a_i is the value of a primitive function, and the coefficients c_{ij} define the descriptive function whose value is d_j. The "decision function" may pick the descriptive function having the highest value, or all such functions having values above some criteria; the program outputs the pattern name associated with the selected function. This



(a) Simple parallel classification paradigm



(b) Simple sequential classification paradigm

Figure 2-1. Simple classification paradigms.

paradigm is parallel in the computation of primitive functions and of descriptive functions; all the primitive functions and descriptive functions are computed, and the order of computation is unimportant.

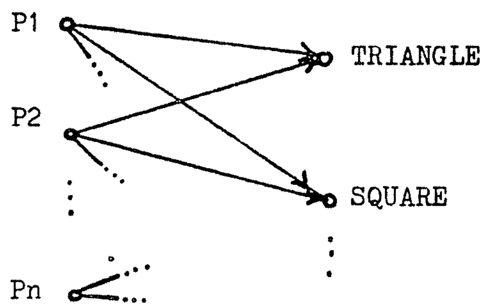
There are literally hundreds of papers describing work which use this paradigm for such applications as recognition of hand printed letters, handwriting, clouds, and bubble chamber events. Dominant themes concern preprocessing techniques (e.g., noise reduction and image enhancement), primitives for use on various pattern classes (feature extraction), and mathematical properties of the paradigm, particularly with respect to learning the coefficients of the evaluation function. Surveys of the work include Nagy [1968], Rosenfeld [1969a] and Rosenfeld [1973]. A more comprehensive treatment is presented in Nilsson [1965], Rosenfeld [1969b], Duda and Hart [1973] and Uhr [1973].

As in the classification technique, HPL uses both linear evaluation functions with learned coefficients and decision functions; they are employed at each memory node to decide when that node has been found. But HPL also computes additional information at each node, including effort expended and likelihood of occurrence, which is used by the recognition algorithm in deciding what step to take next. In addition, HPL's memory is hierarchical, not single-level, as in the most common parallel classification technique.

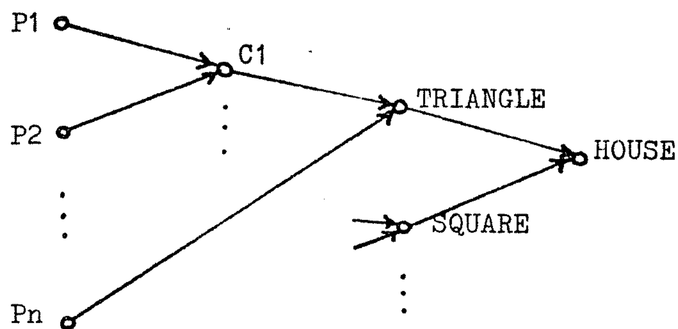
1.11 A Comparison of HPL's Memory and a Single Level Memory

Figure 2-2(a) shows an abstracted form of the memory structure of the simple parallel classification technique. It is a "single level" structure because the primitives P_i link directly in memory to the possible pattern names. Figure 2-2(b) shows an abstracted form of HPL's memory, a multi-level or hierarchical structure in which primitives need not link to pattern names (although they may), but may instead link to intermediate nodes such as C_i, called compounds. Also, pattern names, called goals, may link to other compounds and goals. HPL's more complex structure allows it to describe and recognize patterns in terms of subpatterns, rather than just classifying an input.

Also, HPL's hierarchical memory may save memory space over a single level memory, because information is "chunked" in the psychological sense. Suppose that 20 primitives were used to recognize 1000 words of 5 letters or less. In a single level memory, 20 x 1000 = 20,000 links may be needed. In HPL's memory, individual letters could be represented as memory nodes, and they, rather than primitives, would link to the words. At most 20 x 26 = 520 links would describe the letters, and another 5 x 1000 = 5000 links would describe the words, a total of only 5,520 links. With more complex pattern sets, the saving is difficult to estimate, but the more subpatterns of patterns and subpatterns of subpatterns, etc., the greater the memory saving. With a serial computer, each link requires additional processing; thus, the fewer the links, the



(a) The single level classification memory



(b) HPL's hierarchical memory

Figure 2-2. Abstracted versions of the simple classification memory and HPL's memory.

greater the saving in computer time, as well.

1.12 Uhr and Vossler's Primitive Learning Program

Uhr and Vossler [1961] described an adaptive parallel classification program which learns primitives of a certain specific type (submatrices of 0's and 1's that are swept across the input matrix), extracting them from the inputs that are presented. This is one of the few examples of learning applied to the discovery of primitives. In HPL any primitive satisfying certain properties can be used, but it must be built-in. Uhr and Vossler's technique for learning primitives could be easily added to HPL.

Among the primitives that Uhr and Vossler use are "combinatorial operators", new primitives created from combinations of other primitives. The compound characterizer of HPL is analogous, a combination of characterizers that may be valuable in recognition. Both ideas are approaches toward a similar end consistent with their respective memory structures.

1.13 NAMER, a Relation Recognizer

The NAMER program of Londé and Simmons [1965] ("A pattern-recognition system for generating sentences about relations between line drawings") begins to extend the classification approach into pattern description. NAMER is presented a line drawing of one or two patterns, and outputs a description such as "The square is above, to the left of, and smaller than the circle".

For each pattern, NAMER produces a 96 bit "attribute list" using a set of built-in primitives, e.g., "the number and location of appendages". A similar attribute list and 96 bit "relevance mask" is stored for each pattern in memory. A match score is computed for each pattern as a percentage of bits in the input attribute list which are the same as the corresponding bits in the stored attribute list and which have a corresponding relevance mask bit set to 1. The pattern with the highest score is considered to be the input pattern name. This is a variant of the parallel classification approach with a linear evaluation function. Relevance masks are analogous to the coefficients c_{ij} of the evaluation function and are learned in a similar manner.

The most interesting aspect of NAMER is that relations between two patterns, such as ABOVE and LEFT OF, are learned and recognized in a similar manner, using a separate set of attributes, attribute lists, and relevance masks. In a general sense, NAMER treats relations as another kind of pattern. The output of the pattern and relation naming processes is fed into a simple grammar to produce the final output.

NAMER's recognition ability is limited because of its reliance on a simple classification structure, which does not store structural information about patterns and their relation to subpatterns. For example, although NAMER can recognize the relation ABOVE, that knowledge does not in any way improve its pattern recognition ability. (The Sauvain and Uhr program discussed below, in contrast, does

use explicit relations in higher level constructs.) Nevertheless, storing simple relations in the same memory format (albeit a simple one) as patterns is suggestive, a precursor of Winston's [1970] program in which both relations and patterns are stored within a net-structure (see Section 3.1). HPL does not store explicitly named relations, although it does store implicit part/whole, location and size relations. In Chapter 8, we discuss extending HPL in this important direction.

1.2 The Sequential Classification Technique

In the simple serial or sequential classification paradigm, after each primitive has been computed, a decision is made to either terminate the process and output a pattern name, or to continue; the latter requires deciding which primitive to compute next. Decisions are made on the basis of the outcomes of all the primitives computed so far. Figure 2-1(b) illustrates the basic sequential paradigm. A comprehensive discussion of sequential classification methods is given in Fu [1968].

1.21 The Sequential Technique of Slagle and Lee

Slagle and Lee [1971] propose that the decision tree in a sequential pattern recognition approach may be handled using techniques originally derived for game-playing programs, such as mini-maxing and dynamic ordering of nodes. At each decision point in Figure 2-1(b), the risk (cost) of continuing the process is compared

with the risk of terminating the process and naming a particular pattern. The risk at any node is computed as a "backed-up" value of the risks of nodes connected to it in the decision tree, using the probabilities of the possible outcomes of any primitive function.

The use of cost and value of nodes in HPL's recognition algorithm is conceptually similar to that suggested by Slagle and Lee. The value of a node in HPL, for example, is a backed-up quantity computed as a function of the values of the nodes linked to it in memory.

Slagle and Lee's work is strictly a classification approach and is limited to cases where there are only a small number of features and patterns. As in most of the sequential research, they assume that all relevant probabilities at every level of the decision tree are known, a very limiting assumption, and they make no mention of memory structure. We, on the other hand, make no such assumption about probabilities and are quite concerned with memory structure. All possible probabilities are not stored in HPL, only the most relevant ones, using a mechanism that attempts to discover combinations of memory nodes (called compounds) that make good characterizers. The recognition algorithm of HPL has been designed to work with a memory which is assumed to be inexact and incomplete, for all information is not known by HPL, and the memory would be unmanageably large if it were.

2. Descriptive Techniques of Pattern Recognition

Human perception obviously involves more than assigning names to patterns. Humans can identify subparts of patterns and how they relate to the whole pattern or describe how the pattern relates to the scene of which it is a part. In many mechanical pattern recognition applications, for instance robot vision, a description or structural analysis of the scene is of great importance. Descriptive techniques go beyond just naming a pattern. Evans [1969a] remarks that any classification approach is a special case of a descriptive approach in which the outputs of the primitives are the description, although this description is rarely used as such.

2.1 Syntactic Techniques of Pattern Recognition

Probably the greatest effort in descriptive pattern recognition has been in the application of syntactic or linguistic techniques to pattern analysis. A review of the area is given in Miller and Shaw [1968]. Some of the important issues involved are discussed in Evans [1969a, 1969b]. In brief, a phrase structure grammar with rewrite rules or productions is used to describe a particular class of patterns. The terminal symbols of the rules are what we have called primitives. The non-terminal symbols, or at least some of them, correspond to parts of the pattern. Parsing the input pattern, a set of terminal symbols, is the process of recognition, and the resulting parse is the description of the pattern.

Frequently, the pattern is preprocessed before the primitives are applied.

It is difficult to find pattern sets sufficiently well-structured to be described by a grammar. Consequently, applications of this technique have been limited to certain restricted pattern sets, including chromosomes, fingerprints, and letters of the alphabet. It may be quite difficult to find grammars general enough to process a variety of pattern sets.

The syntactic approach changes the emphasis of recognition from classification to description. However, the present conception of a grammar seems overly restrictive. For example, matching the left hand side of a grammatical production generally requires an exact match of all its components. If the production indicates where components are located, they must be located exactly as specified for the rule to succeed. Real world patterns often do not exhibit such nice properties. It is unlikely that all problems of inexactness can be handled by preprocessing. It is possible that grammars can be generalized to allow for probabilistic, inexact, or incomplete matches. Evans [1969a] suggests combining statistical classification techniques with the syntactic approach, using the former to determine the presence of constituents, which would then match parts of grammatical rules. Since a central problem in syntactic pattern recognition is describing a suitable grammar, it might be of great use to develop techniques for learning grammars. Grammatical inference, however, is still in its

early stages. (See Feldman, et al. [1969].)

The central memory construct of HPL, the characterizer, may be thought of as a very loose, flexible type of grammatical rule. However, HPL's recognition algorithm is not based on any conventional parsing algorithm, and so it is probably unwise to stretch the analogy too far.

Guzman [1971] discusses a syntactic approach to the recognition of cartoon drawings and how the "context problem" may be handled by parsing and backtracking within this framework. We contrast this approach with our suggested approach in Chapter 8.

2.12 The Sauvain and Uhr Program

A precursor of HPL is the adaptive pattern description program of Sauvain and Uhr [1969] (which we shall call SU). One of our aims has been to achieve the goals of SU in a more general framework.

A line drawing is presented to SU as lists of numbers representing the length, slope, curvature, and location of lines in the pattern, the primitives of SU. SU outputs the names of patterns and subpatterns in the input and relations between them, for example, "HOUSE CONSISTING OF TRIANGLE ABOVE SQUARE".

SU's memory is a set of "inference rules" of the form:

OBJECT1 S-RELATION OBJECT2 = NEWNAME.

OBJECTi may either be a primitive or a name appearing in the NEWNAME field of some other rule. NEWNAME is a renaming of either

the entire left-hand side of the rule or only of OBJECT1, the remainder of the left-hand side serving as context. Each OBJECTi is enclosed within a horizontal rectangle. S-RELATION specifies the relative location of the rectangle enclosing OBJECT1 with respect to the rectangle enclosing OBJECT2. It is represented as a list of number pairs, e.g., [2-3,0-99,...], meaning that the difference between the leftmost coordinates of OBJECT1 and OBJECT2 must be at least 2 and at most 3, etc. One or more S-RELATIONs may be given a name, such as ABOVE or BELOW. Inference rules and S-RELATIONs are created when feedback is given by a trainer.

SU and HPL both attempt to describe a pattern in terms of sub-patterns. SU appears to have a greater descriptive ability, for it can output TRIANGLE ABOVE SQUARE, whereas HPL would output TRIANGLE, its location and size, and SQUARE, its location and size. SU's relation ABOVE, however, is just a renaming of a particular S-RELATION, a comparison of relative locations. HPL implicitly uses relative location (and size, as well) within its basic memory unit, the characterizer, although relative location is not part of the output. SU's ability to name relations meaningfully would be more powerful and significant if general relations could be learned from lower-level relations (e.g., allowing new relation names on the right-hand side of an inference rule). HPL's "associated rectangle" comes directly from the similar concept in SU, although generalized to include size.

The memory of both HPL and SU is hierarchical. However, HPL's characterizer is more general than SU's inference rule. It contains any number of components; furthermore, weights are associated with components and thresholds with characterizers, allowing HPL to recognize patterns of greater complexity than SU. In addition, SU assumes a particular kind of primitive; HPL allows any primitive meeting certain requirements. We believe HPL's structure is more amenable than SU's to extensions.

SU learns through several types of explicit feedback, indicating, for example, that a particular pair of patterns in a certain S-RELATION should be renamed, or that a particular pattern should be renamed in the context of the other patterns present. Feedback in HPL is intentionally less specific, for we feel that the less required of a trainer, the better. Names of patterns and their locations, YES and NO, are the only kinds of feedback HPL uses.

SU's use of a special context-sensitive inference rule is interesting, although dependent on giving exactly the right feedback. In Chapter 8, we suggest a method of handling context in HPL without defining a special context-sensitive characterizer.

SU's recognition algorithm finds all possible inference rules that might apply to the input, starting with the primitive strings. HPL's algorithm is considerably more complex, attempting to search memory and compute primitives in an efficient order. It does not necessarily compute all primitives or access all potentially relevant characterizers.

2.2 Scene Analysis Programs

The dominant trend of current scene analysis programs has followed the lead of Guzman's [1968] program. A good summary of the MIT work in this area is presented in Winston [1972]. The main concern of this work is segmentation of possibly unknown objects in a scene, usually two-dimensional line drawings of geometric solids, rather than object identification. The programs have a syntactic flavor (e.g., labelling components of scenes) but have generally not used explicit linguistic rules. Except for Winston [1970], these programs have not used learning techniques.

Guzman [1968] segments line drawings of scenes of geometric solids by identifying vertex types and linking regions together using the vertex information and heuristics concerning the occurrence of parallel lines, lengths of edges, etc. The heuristics are grouped into "strong" and "weak" evidence, with the former directing the process.

The most significant extension of Guzman's work is that of Waltz [1972], which segments scenes of geometric objects containing shadows. His work is more powerful, simpler, and less *ad hoc* than Guzman's. As in Guzman, identification of basic vertex types is important. The program attempts to "label" the lines in the drawing according to their illumination and physical cause, e.g., a line may be a shadow or the intersection of two visible surfaces. Using a "filtering" technique which compares the information that adjacent

vertices imply about the labelling of particular lines, Waltz is able to efficiently label most lines by eliminating conflicts. Similarly, labels are generated for regions, depending on the information stored with line labels. Waltz's program is both efficient and effective on fairly complex scenes.

HPL's simple scene analysis ability is not directly comparable with that of the work just cited. The Waltz program, in particular, segments scenes including shadows of much greater complexity than HPL can handle. However, unlike HPL, the Waltz program does not attempt to recognize objects, nor does it learn.

The Guzman and Waltz programs are strictly "bottom-up". uilt-in knowledge about vertex types and their implications in combination with the information in the input scene are used to generate a scene description. HPL is neither strictly top-down nor bottom-up, using information from the input and what is <u>expected</u> about the input to drive the recognition process.

Falk [1971] is interesting in this regard, for he gives his Guzman-style scene analysis program a slight top-down flavor. Objects are recognized, following segmentation. The recognition itself is rather simple, for only a small number of pattern types are allowed; sets of features associated with named patterns are compared against features actually found. Following recognition, a scene is generated and compared with the input. Significant discrepancies direct the recognition process to re-recognize an object. This, however, is limited to assigning an alternate name

to an object where more than one name was assigned by the recognition process initially. The idea is appealing, however, if extended more generally to affect details of the recognition process (as expectation directs recognition in HPL).

The Guzman type of approach and our rather different approach are not necessarily completely mutually exclusive. For example, in Chapter 8, we suggest how some vertex information may be built into HPL's primitives.

3. Net-Structure Approaches to Pattern Recognition

In the classification paradigm, an internal description of each possible pattern is implicit in the combination of primitive and descriptive functions pertaining to that pattern. In syntactic approaches, the internal description of patterns is contained in the grammatical productions. In what we call net-structure approaches, the internal description is contained within a net-structure memory, that is, a memory in which nodes are implicitly or explicitly linked to other nodes.

3.1 The Winston Program

Although Winston [1970] may be considered an extension of the Guzman type of scene analysis program, his concerns are closer to our own, including pattern description, recognition and learning with an explicit net-structure memory. Winston applies various heuristics to the output of Guzman's scene analysis program to

produce a net-structure description of objects within the scene. Links between nodes express relations between subparts of objects and between whole objects, for example, "supported-by", "a-kind-of" and "has-property-of". A column consisting of three cubes sitting on top of each other might be represented as a graph with cubes as nodes and "supported-by" links connecting them. Pattern concepts are formed by comparing net-structure descriptions of two different instances of a concept to form a more general net-structure description of the pattern. Recognition is accomplished by comparing previously stored descriptions with the description generated for the current pattern. While the descriptive ability of Winston's program is impressive, its recognition ability is time-consuming and cumbersome. HPL differs from Winston's program in several basic respects. A comparison of HPL with Winston's program raises some important issues, which are discussed in detail in Chapter 9.

3.2 Pattern Recognition Techniques of Uhr

HPL has been greatly influenced both in structure and philosophy by the work of Leonard Uhr, summarized in Uhr [1973]. Uhr has extended the classification paradigm in several continuing and important directions and is greatly concerned with learning and memory structure. His extensive work is difficult to summarize, for he emphasizes the great range of possibilities inherent in his net-structure programs.

The basic unit of Uhr's memory is called a characterizer, as is the similar (but far from identical) construct in HPL's memory. As typically defined, a characterizer contains three components, a description component, an implication component, and an action component.* The description component literally "describes" the characterizer in terms of other characterizers. It is a list of items of the form:

[characterizer name, location, description weight].

The implication component indicates pattern names "implied" if this characterizer is found. It is a list of items of the form:

[pattern name, threshold, implied weight].

The action part indicates what characterizers to "look for" if this characterizer is found. It is a list of items of the form:

[characterizer name, location].

Recognition proceeds as follows: For every characterizer, a tally is kept of the sum of the description weights of characterizers in its description component which have already been found. If that sum exceeds the threshold of an implied pattern name in the implication component, then the characterizer itself is added to a list of characterizers which have been found, and the implied pattern name is added to a list of possible pattern names, along with its implied weight. Characterizers in the action component are added to a list of characterizers to be looked for. When the

---

*The terminology used here is largely our own and examples are in LISP-like list notation rather than the notation used by Uhr.

cumulative implied weight of a pattern name exceeds some fixed criterion, then it is output by the program. Feedback in the form of YES, NO, or the appropriate pattern name causes the description weights and the implied weights to be adjusted.

This basic idea is subject to many variations in Uhr's work, particularly with respect to learning, generality, and flexibility. Learning may be inserted into this structure at many points. For example, in Uhr and Jordan [1969], characterizers are generated, rather than being built-in. Generality is stressed in relating pattern recognition to concept formation and language learning. "Flexibility" is a term used by Uhr connoting ways to allow for variability and inexactness in determining when a characterizer is found.

Uhr typically describes a set of programs, or more accurately, prototypes of programs, emphasizing the many variations that can be made in them. We, in contrast, have developed a detailed, extensive program, concentrating on specific problems.

Uhr's general memory structure has served as the starting point for HPL's structure, and there are basic similarities between them. HPL's memory is also made up of characterizers, which contain a description component and an implication component, but no action component. (HPL's recognition algorithm uses the implication component as if it were an action component, indicating characterizers to look for.) HPL uses the description weight of Uhr (which is also similar to the coefficient c_{ij} of the linear evaluation function

used in parallel classification programs). Many of the differences between HPL's memory and Uhr's memory arise because of HPL's sequential recognition algorithm. For example, HPL's use of implication weights (a measure of the probability between nodes), cost, value, and other parameters associated with memory nodes is not part of Uhr's structure, nor is the use of active memory and instances. Extensions of HPL discussed in Chapters 8 and 9 diverge to an even greater extent from Uhr's approach.

3.3 Becker's Model of Intermediate Level Cognition

Becker [1970] proposed an adaptive net-structure model of "intermediate level cognition" called JCM. It is more akin to robot and simulated robot work than it is to pattern recognition (which he barely mentions), and so we shall discuss the work only briefly.

JCM's basic unit of long term memory is the "schema", which has the form:

[EVENT1 IMPLIES EVENT2],

where each EVENTi is a list of "kernels" ordered in time. Each kernel has the form:

[FN ARG1 ARG2 ...],

where FN can either be the name of a primitive (e.g., a motor command such as "move hand up one unit") or a "secondary kernel", which is a method of naming schemata. Weights are used abundantly in JCM; for example, there are weights associated with kernels and

arguments of kernels. Except for the time ordering, a schema is similar to the characterizer of HPL (and of Uhr), and kernels resemble the method used to reference characterizers in HPL.

In JCM, secondary kernels naming schemata are used to build a hierarchical structure. However, the process by which JCM creates such kernels is admittedly vague. In contrast, the creation of a hierarchical structure is essential to HPL's operation. All characterizers have names and may participate in higher level characterizers, and a specific mechanism is described that decides which characterizers actually participate in higher level characterizers.

An important process in JCM is "schema-application", a heuristic search method used to determine what kernels and schemata the model will process next. Although HPL's recognition algorithm is quite different than Becker's schema-application process, they both share the important concept of treating memory as a graph and using heuristic search techniques to process it.

CHAPTER 3

A GENERAL DESCRIPTION OF THE PROGRAM HPL

1. An Introduction to HPL

In this chapter we describe the organization and operation of HPL in general terms. In the following three chapters, we discuss HPL in more detail. This chapter serves as a necessary introduction to those chapters. We first describe the net-structure memory of HPL and its components, then the processes of HPL, and finally the context problem.

HPL consists of a net-structure memory and the processes that operate on that memory. The memory greatly influences the form of the processes of HPL. It is a unified structure, built up from a single type of unit. HPL's memory is learned; its processes, the recognition process and the learning process, are built-in. The learning process follows the recognition process. The recognition process could function alone in recognizing patterns if HPL were provided with a built-in memory, but this was not desired.

HPL's memory consists of three parts, long-term memory (LTM), intermediate-term memory (ITM), and active or short-term memory (STM). (The cognitive model flavor of these names is meant to be suggestive but nothing more.) The basic unit of memory is the characterizer. LTM is the reservoir of information retained over time through learning. ITM is structurally identical to LTM, but serves a special purpose in the learning process. It is the

storage area for new characterizers created by the program, which may eventually be transferred to LTM if they are "good" enough. STM is a temporary storage area used to store the information in LTM in a more specific form applicable to the current input scene. STM instances are created of LTM and ITM characterizers. One LTM characterizer may correspond to several different instances. The instance contains all of the information accumulated during the recognition process. The current state of STM helps determine what step HPL will take next in the recognition process. STM changes continuously during recognition. In the learning process, STM changes no further but is kept as a record of what occurred during recognition. The information in STM in conjunction with feedback determines what learning will take place.

The input to HPL is one or more patterns given in a format which depends on the type of primitives used. With submatrix primitives, the input is an M x N matrix of 0's and 1's. With line primitives, the input is a list of vertices and their x,y coordinates (see Chapter 7). The input does not have a time component; that is, inputs are isolated snapshots. We will often call the input a "scene" whether we are speaking about one pattern or more than one. A "frame" is the following sequence of events: A scene is presented to HPL. HPL recognizes patterns in the scene as best it can and is given feedback during the recognition process as to the correctness of its responses. After the

recognition process has terminated, the learning process modifies the memory using the information accumulated during recognition, including feedback.

2. Information Flow Diagrams of the Processes of HPL

An information flow diagram of the recognition process is shown in Figure 3-1. Arrow 1 indicates that the only communication between the input scene and the rest of the program is via the primitive functions, which compute properties of the input. Thus, there is an intimate connection between the form of the input and the form of the primitives. The results of primitive evaluation are sent to the recognition process (arrow 2).

Arrow 3 in the opposite direction is of particular importance, for it indicates that the recognition process controls the evaluation of primitives by deciding what primitive to compute next. In a parallel system in which all possible primitives are computed simultaneously, this arrow would not exist. (It would exist, however, in a combination parallel-serial system.)

The recognition process accumulates information in STM, altering its structure as more is learned about the input (arrow 7). The information in STM, in turn, is used during recognition to help decide what primitive to compute next and what patterns are present in the input (arrow 6). The recognition process determines what information is transferred from LTM and ITM to STM (arrows 5 and 8). Although most memory information comes to the

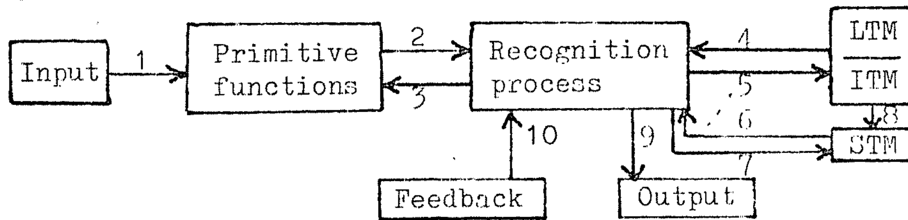


Figure 3-1. An information flow diagram of the recognition process. Numbered arrows correspond to the description in the text.

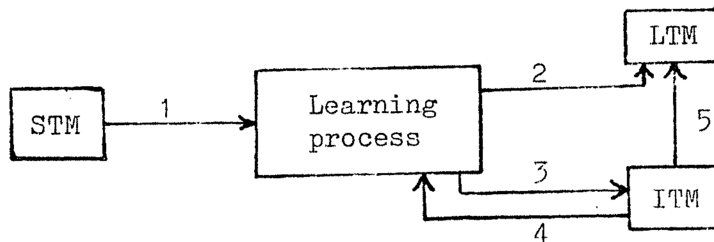


Figure 3-2. An information flow diagram of the learning process.

recognition process via STM (arrow 6), a small amount comes directly from LTM and ITM (arrow 4). This arrow could easily have been eliminated by incorporating the information into STM, but with a corresponding increase in the memory size of STM.

The end result of recognition is HPL's response, the names of the patterns in the scene (arrow 9). After HPL responds, feedback external to the program may optionally be given indicating its correctness, and providing the names of any patterns in the scene that should have been output but were not (arrow 10).

An information flow diagram of the learning process is shown in Figure 3-2. The information accumulated in STM during recognition, including feedback, is used by the learning process to determine what learning shall occur (arrow 1). The learning process modifies the permanent memory, LTM and ITM (arrows 2 and 3). Information in ITM is occasionally analyzed to determine whether it should be permanently retained and transferred to LTM or whether it should be discarded from memory completely (arrows 4 and 5).

3. The Memory Structure

The basic unit of memory is called a _characterizer_ because it "characterizes" or describes some "meaningful" part of a pattern. Characterizers are of three types: _primitive_, _compound_, and _goal_. They have the same form but are treated in different ways.

Primitives are built-in functions which compute properties of

the input. They are the simplest characterizers, for they are not described in terms of any other characterizers. Primitives may represent straight line segments of a certain slope or M x N submatrix templates of 0's and 1's.

Compounds and goals are described in terms of other characterizers of any of the three types. They are identical in structure but are distinguished in the following way. Each goal represents a unique pattern name; that is, there is a one-to-one correspondence between goals and pattern names. Whenever a new pattern name is given to HPL in feedback, e.g. HOUSE or FACE or LETTER-W, a goal characterizer will be created to uniquely represent that name. Recognition is the process of deciding which goal characterizers are present in the input; when HPL finds a goal in the input, the pattern name associated with it is output as a response. Compounds in contrast do not represent anything defined externally by the trainer; they are formed internally in the hope of discovering combinations of characterizers which are more valuable in describing patterns than the individual characterizers by themselves. For example, a compound representing two straight lines meeting at a right angle may be a more "valuable" characterizer than either line alone. (The meaning of "value" is explained in the next chapter.)

Every characterizer has an internal name by which other characterizers may reference it. We will use these names when

discussing characterizers: primitives will be called P1, P2, etc., compounds C1, C2,..., and goals G1, G2,.... When some other letter such as A or X is used, we will usually mean a characterizer of any type. Occasionally, we will use a meaningful name for a primitive (e.g., LINE-4 represents a straight line having a discrete slope of 4), but this is irrelevant to HPL's operation. More frequently, since every goal has an external pattern name associated with it, such as CAT or FACE, we may refer to the goal by that name rather than its internal name.

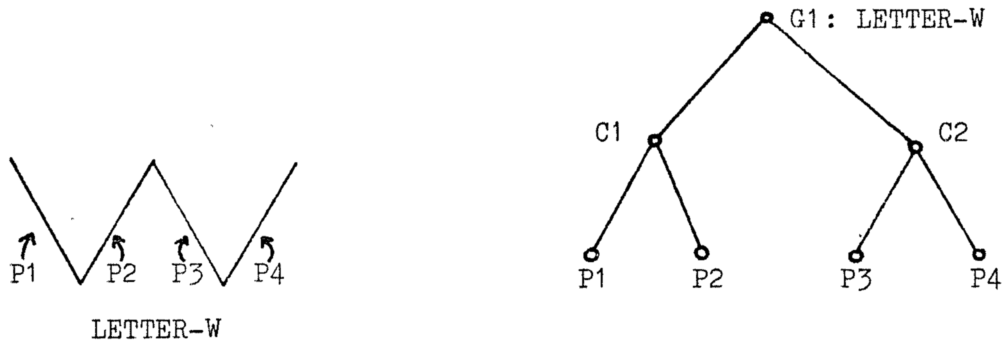
Figure 3-3 illustrates HPL's memory structure as a graph*. How the graph is explicitly stored within characterizers is explained in the next chapter.

In Figure 3-3(a), each straight line in the pattern LETTER-W is represented as a separate primitive; for example, P1 is the left-most straight line in the pattern. Primitives are terminal nodes in the graph. Compounds and goals are non-terminal nodes, defined (i.e., described) by their successor nodes. For example, if P1 and P2 are both found** in the input, then C1 is found as

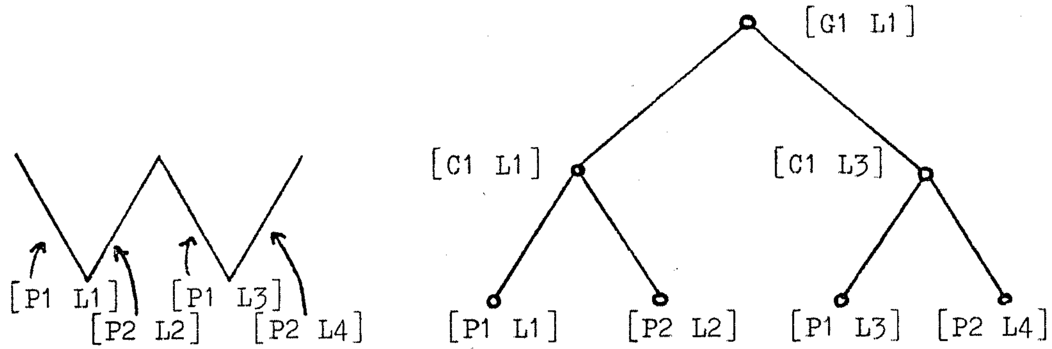
---

*In the graph representation of memory, characterizers are nodes. Connections between nodes are "links". Downward links from one node to another are "successor" links, upward are "parent links". "Upward" and "downward" refer to a graph oriented so that primitives are at the bottom. A node X is "higher-level" or "lower-level" with respect to node Y as there is a downward or upward path from X to Y.

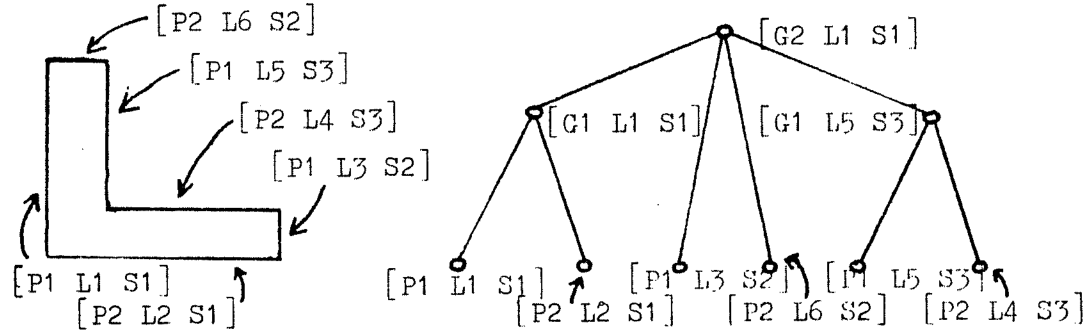
**When HPL decides that a characterizer is present in the input, we shall say it is "found in the input" or simply "found".



(a) Individual lines as primitives



(b) Primitives as lines of a particular slope



(c) P1 and P2 are vertical and horizontal lines. There are six locations L1 through L6 and sizes S1 through S3. Goal G1 is a CORNER, G2 a BLOCK-L.

Figure 3-3. The memory structure of HPL represented as a graph.

well. Similarly, if both C1 and C2 are found, then G1 is found, and since G1 is a goal, the external name associated with it, LETTER-W, will be output. In this example, the highest-level node is G1, which has no parents. This is not usually the case, for goals may describe higher-level compounds or goals.

From this example, it would appear that a compound or goal is a logically AND'ed combination of its successors; hence, the graph could be represented by syntax rules such as

"P1 AND P2 = C1", "C1 AND C2 = G1", etc.

This is a simplification, for the links actually have weights associated with them, as well as a threshold at each node. The weights are bidirectional; there are separate weights for upward and downward links, called the description weight and the implication weight. The description weight determines the importance of a node in deciding that its parent is found. The implication weight measures how likely a node is, given a successor node.

We shall now move a step closer to HPL's actual memory. Many primitives would be necessary to represent every straight line in every possible location; memory would be very large, since each characterizer requires memory storage. To handle this problem, a primitive is a function that computes a property at a location which is given to it as an argument, rather than a function that computes a property at a _fixed_ location. For example, in Figure 3-3(b), LETTER-W is represented by two primitives, each at

two different locations, [P1 L1], [P1 L2]* etc., rather than four separate primitives. Similarly, a location may be associated with any characterizer: [CHARACTERIZER-NAME, LOCATION]. For a non-primitive, this location can be thought of as the location of its leftmost successor. (The actual relation of the location of a non-primitive to its successors is somewhat more complicated and is explained in Chapter 4.) The graph of LETTER-W now contains only one compound C1 at two locations, rather than two compounds. [G1 L1] is at the top of the graph, so if it is found, HPL can output its location as well as its name.

In HPL's memory representation, size is also associated with a characterizer, effecting an additional saving in memory space, and allowing HPL to include size in its output. This is represented by: [CHARACTERIZER-NAME, LOCATION, SIZE]. (Additional arguments not presently implemented in HPL are discussed in Chapter 8.) Figure 3-3(c) illustrates the use of size. The straight lines meeting at a right angle are called a CORNER. The pattern BLOCK-L contains two CORNERs as subpatterns at two locations with two different sizes. This example also illustrates that a goal may have parents.

Figure 3-3(b) introduced location as an argument of a characterizer. How is location stored in memory? If [C1 L1] and [C1 L3] are stored as separate nodes in permanent memory, then

*For readability, we shall use parentheses or brackets interchangably to designate lists.

there is no saving of memory space over storing the separate characterizers C1 and C2 of Figure 3-3(a). In order to effect a saving, only one node representing C1 is stored in memory. C1 is defined in memory in terms of its successors P1 and P2, and their relative locations and sizes with respect to each other. Eventually, these relative attributes must be related to absolute attributes indicating actual locations and sizes in the input.

The short-term instances of long-term characterizers which are created during recognition contain absolute location and size. For example, in Figure 3-3(b), C1 appears twice in the graph. The structural definition of C1 occurs only once in permanent memory; however, during the recognition process, two instances of C1 will be created, one for each absolute location. In fact, an instance will be created for each node in the graph.

4. The Recognition Process

The recognition stage attempts to access memory and compute primitives efficiently, using learned information, with the overall aim of determining what goal characterizers are present in the input. Recognition proceeds by finding characterizers in the input (nodes in the graph), moving upward in memory to their parent nodes, and downward from the parent nodes to successor nodes which have not yet been found. The order in which nodes are accessed depends on several factors (see Chapter 5), but in

general, HPL tries to investigate characterizers that seem likely to succeed, are not extremely costly, and are valuable in providing information about the occurrence of goals. There are two main sources of this information: the weights associated with the links between characterizers in LTM and the record accumulated in STM during the course of recognition as to what success HPL has had in finding characterizers which describe higher-level characterizers. For example, if some characterizer X is described by characterizers A, B and C, and A has been found and B has not, then there is less chance that HPL will investigate C than if both A and B had been found.

After HPL responds with the name of some goal it has found, feedback may be given indicating "YES" or "NO" or the correct name of some part of the pattern either misrecognized or not recognized at all. The feedback affects further recognition immediately. For example, if HPL responds "SQUARE", and feedback is "NO", HPL will not continue to look for patterns containing a SQUARE. Information indicating whether or not an instance is found is stored within the instance and is accessible by the learning process.

We shall give an example of how the recognition process works, postponing a detailed explanation to Chapter 5. Suppose HPL is given the pattern shown in Figure 3-3(b) and has already found P1 at location L1, i.e., the leftmost straight line. Finding P1 may cause all parents of P1 in memory, that is, all characterizers

whose descriptions include P1, to be accessed; instances will be formed of each parent and put on a list called the ACTIVE list. C1, representing the V shape consisting of P1 and P2, is such a characterizer; an instance of it will be formed at absolute location L1, and also put on the ACTIVE list. HPL "investigates" the first instance on the ACTIVE list, removing it from the list. When C1 is investigated, the characterizers which describe it, namely P1 and P2 at the proper locations will be inspected. HPL will realize that P1 has already been found; but since P2 at location L2 has not been found, HPL will form an instance of it. There is then a chance that P2 will be investigated; since P2 is a primitive, the function it represents will be computed, that is, a straight line of the proper slope will be sought at location L2. After P2 is found, HPL will decide that C1 is found, since both its successors have been found. This may in turn cause parents of C1 to be accessed. In a similar manner, instances will be created of [G1 L1], then [C1 L3], then [P1 L3], etc. When [P1 L3] and [P2 L4] are eventually found, then [C1 L3] and finally [G1 L1] will be found, and at that point the pattern name LETTER-W associated with G1 will be output. The formation of instances of successors of nodes is analogous to creating subgoals as steps in the process of achieving some higher level goal. A record of these subgoal/goal links is maintained within the instance.

5. The Learning Process

The learning process adjusts the weights associated with links between characterizers in permanent memory (LTM and ITM), creates new links, forms new compound characterizers, and analyzes compounds to determine if they are worth saving.

There are two kinds of weights stored in a characterizer. One indicates the importance of a characterizer within the description of a higher-level characterizer; the other is the relative probability that having found a certain characterizer we may expect to find its parents. The weights are adjusted upward or downward depending on which characterizers have been found and which have not. For example, if the description of HOUSE contains a rectangle, and no rectangle was found in a HOUSE presented to HPL, the weight associated with rectangle in the description of HOUSE will be lowered. If the weights reach zero, links will be discarded. Weight adjustment, although rather straightforward, is very important. Other sections of the learning process are more problematic, and our approach was more experimental.

One part of the learning process decides when to create new links between characterizers. Should links be established between a new goal or compound and every other characterizer in memory? This is inadvisable because a very large number of links would have to be handled (although if they could be handled, the weight adjustment process would, in time, weed out the least

valuable links). Our algorithm (see Chapter 6) picks a few new links to be formed at appropriate times.

Goals are created because of external feedback, but compounds are created internally. They attempt to represent valuable combinations of lower-level characterizers. We are confronted with a problem similar to the one just mentioned. Should compounds be formed for every possible combination of lower-level characterizers? This would create a large number of compounds, although the compound analysis procedure would theoretically, over time, weed out bad compounds. Again our approach has been to sample some but not all possible compounds. Such compounds are placed into ITM. After a period of residence in ITM, compounds are assessed as to their value in contributing to recognition. If certain criteria are met, compounds are transferred to LTM, otherwise they are discarded.

6. The Context Problem

It may not be possible to uniquely name a particular pattern by its shape alone; the environment in which that shape is found, the "context" associated with that shape, may be necessary to resolve the ambiguity in naming the pattern. This is the "context problem". In the idealized patterns HPL handles, a circle, or even a square, might represent an eye in the context of a face, but a

wheel in the context of a wagon. We sought an approach to this problem which would not require great modification to the existing memory structure. A new weight associated with links between characterizers was introduced which is treated much like other weights. Our approach to context has not yet been implemented in a running version of HPL, and is discussed along with other extensions in Chapter 8.

CHAPTER 4

THE MEMORY STRUCTURE

1. The Characterizer

The characterizer is the unit of information storage in permanent memory, i.e., LTM and ITM. (Unless we are discussing the learning process, we shall refer to LTM loosely, including both LTM and ITM.) It corresponds to a node in the graph representation of memory used in the previous chapter. The instance, the unit of short-term memory storage, is described in Section 2.

Every characterizer has a unique internal name, the characterizer name, which other characterizers use in referring to it. The stored information which defines a characterizer is only accessible from its name. (In LISP, this information is stored on the property list associated with the atomic symbol specifying the name.)

The components of a characterizer which describe the net-structure are the description-set and the implication-set. The description-set specifies the conditions that must be satisfied to determine if a characterizer is found in the input. If a characterizer is found, its implication-set specifies what characterizers to expect. The description-set and implication-set are analogous to a premise and conclusion. The structures of the description-set and implication-set are identical but are

interpreted differently. In addition to the description-set and implication-set, several additional items are associated with each characterizer name. These components will now be described in detail.

1.1 The Description-Set

The description-set of a characterizer is a list of descriptors. Each descriptor is a link between the characterizer and other characterizers (successor nodes in the memory graph). The descriptors of a characterizer define it. When enough descriptors have been found (as explained in Section 1.13), then the characterizer itself is considered found. Primitives are the only characterizers with no description-set, for they are not defined in terms of other characterizers. They are considered found if the function defining them succeeds.

The form of a descriptor of some characterizer X is:

[(NAME LOC SIZE) DWT].

(NAME LOC SIZE) is called a characterizer reference, for it refers to a characterizer NAME, having relative location LOC and relative size SIZE with respect to X. DWT, the description weight, indicates the importance of this descriptor in describing X, i.e., in determining when X is found.

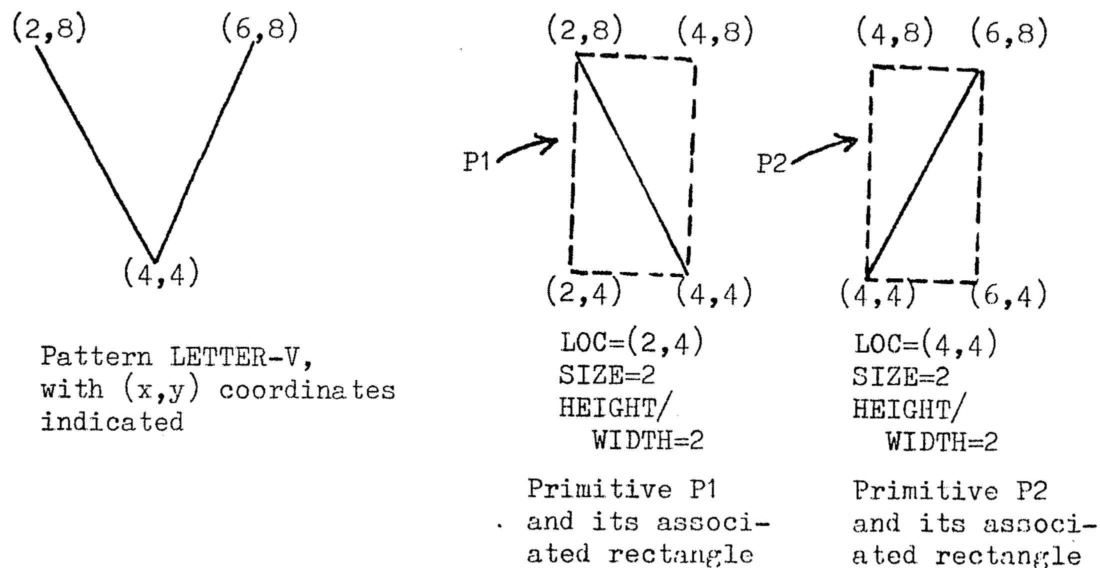
1.11 The Associated Rectangle, Location and Size

An implied rectangle is associated with every characterizer, called the associated rectangle. It is the smallest horizontal rectangle that completely surrounds the characterizer; for a non-primitive, this means the smallest horizontal rectangle surrounding the associated rectangles of its descriptors. LOC and SIZE in a characterizer reference signify the location and size of the associated rectangle.

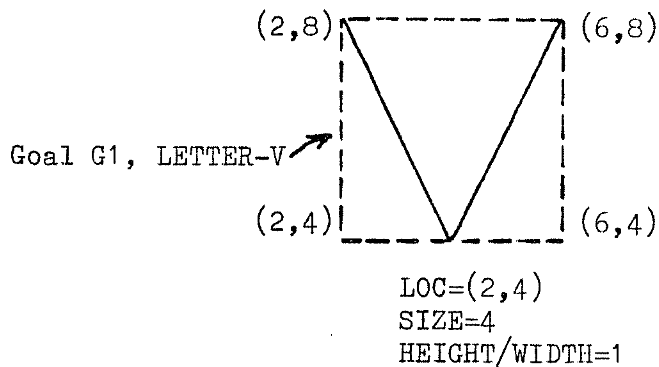
The associated rectangle is specified by location, height to width ratio, and size. Its location is the (x,y) coordinate of its lower left-hand corner. The height to width ratio is the ratio of the length of the vertical side of the rectangle to the horizontal side, and is stored as one of the additional items associated with a characterizer name. Size is the length of the horizontal side of the associated rectangle. Figure 4-1(a) shows the rectangles associated with two line primitives, P1 and P2, and Figure 4-1(b) shows the associated rectangle for the goal G1 (LETTER-V) composed of the two primitives.

1.12 Relative and Absolute Attributes

Location and size as stored within the description-set (and implication-set) of some characterizer X are relative attributes, specifying the location and size of descriptors with respect to the location and size of X. During the recognition process, these relative attributes are related to absolute attributes by matching



(a) Rectangles associated with primitives P1 and P2



(b) Goal G1 composed of P1 and P2, and its associated rectangle

Figure 4-1. Associated rectangles.

instances to descriptors and implicands (see Section 2.3). Figure 4-2 shows the pattern LETTER-V represented by the primitives [P1 (2,4) 2.] and [P2 (4,4) 2.], where location and size are absolute quantities. Descriptors of G1, however, specify the relative locations and sizes of P1 and P2 with respect to G1. The relative location of P1, for example, is (0,0). This indicates that if G1 is found at absolute location (x,y), then P1 must also have been found at absolute location (x,y).

More generally, suppose a characterizer has been found with absolute location (x,y) and size s and has a descriptor with relative location (x_r, y_r) and size s_r. The absolute location (x_a, y_a) and size s_a of that descriptor are computed as follows:

$$x_a = x + x_r \cdot s, \qquad y_a = y + y_r \cdot s, \qquad s_a = s \cdot s_r$$

1.13 The Description Weight

DWT, the description weight of NAME in the description-set of X, indicates how important NAME is in describing X. During the recognition process, HPL looks for descriptors. If NAME is found at LOC with SIZE, then DWT, a non-negative integer, is added to an accumulated sum, the cumulative description weight (CUM-DWT), associated with an instance of X. When this sum exceeds the THRESHOLD of X, then HPL concludes that X is found in the input. The description weights and THRESHOLD are initially set when a characterizer is created or structurally modified and are adjusted in the learning stage.

1.14 Weighted Links with Threshold for AND and OR Operations

A characterizer is found when the sum of the description weights of its descriptors that have been found exceeds a threshold. HPL has no AND'ed or OR'ed combinations of links, only weighted links. Weighted links, however, are more general than AND/OR links. For example, if the THRESHOLD of a characterizer equals the sum of the description weights of its descriptors, then in effect the description-set is an AND'ed combination of descriptors; all must be found for the characterizer to be found. If THRESHOLD equals the description weight of one descriptor, and if all description weights are equal, then the description-set is an OR'ed collection of descriptors. Intermediate values of description weights describe more logically complex conditions. For example, suppose some characterizer has a THRESHOLD of 10 and four descriptors, a, b, c, and d, with description weights of 5, 5, 3, and 3, respectively. The characterizer will be found if either a and b, or a, c, and d, or b, c, and d are found.

A disadvantage of weighted links is that they are not interpreted as easily as AND/OR links. A significant advantage, however, is that a weighted structure is especially amenable to learning. Weights may be adjusted in small increments with a cumulative effect approximating an AND or OR link. Learning a structure with explicit AND/OR links is more difficult. Forming a link is an act of much greater consequence than a small adjustment of weights. (We compare one learning program that does use

explicit links [Winston, 1970] with HPL in Chapter 9.)

1.2 The Implication-Set

The implication-set is a list of _implicands_. Each implicand, like a descriptor, is a link between the characterizer and other characterizers, but, unlike the descriptor, is a link to parent nodes, rather than successor nodes. An implicand of some characterizer X is a characterizer which may be expected if X is found.

The form of an implicand of some characterizer X is:

[(NAME LOC SIZE) IWT].

The characterizer reference (NAME LOC SIZE) is identical in meaning to the characterizer reference in the descriptor; LOC and SIZE are both relative with respect to X. IWT, the _implication weight_, indicates the probability of this implicand, having found X.

1.21 The Implication-Weight

IWT is a non-negative integer reflecting the probability of finding (NAME LOC SIZE), given X. This probability is not IWT itself but is easily obtained from it by dividing IWT by USE, one of the additional items associated with NAME. USE is the number of times NAME has been found in prior inputs. As discussed in the next chapter, IWT figures prominently in the recognition process.

1.3 The Relationship between Descriptors and Implicands

Descriptors and implicands are closely related. If

characterizer A is a descriptor of characterizer X then it must follow that X is an implicand of A. For example, if SQUARE is a descriptor of HOUSE, then HOUSE, in turn, is an implicand of SQUARE. This means that links between characterizers are bidirectional.

The implication-set could theoretically be eliminated, incorporating IWT into the description-set, e.g.

[NAME (LOC SIZE) DWT IWT],

where DWT is the description weight of NAME with respect to X, and IWT is the implication weight of X with respect to NAME.

The implication-set is important, however, in terms of efficiency. During recognition, when a characterizer X is found, its implication-set determines which characterizers are expected. If it were possible to efficiently search memory to determine what characterizers contain X in their description-set (e.g., with a content-addressable memory), the implication-set could be eliminated as described above. Explicit storage of the implication-set, then, saves computer time but uses additional permanent memory space.

1.4 Additional Information Associated with a Characterizer Name

There are several additional items stored in memory and associated with each characterizer name but not part of either the description-set or implication-set. We shall enumerate them here, but will elaborate on several of them when describing the recognition and learning processes.

The TYPE of characterizer X indicates whether it is primitive,

compound or goal.

The PRINTNAME of a goal is the external name which HPL outputs when the goal is found. It is learned as a result of feedback.

INSTANCES is a list of all instances of characterizer X created during the recognition process. At the end of the learning process, this list is erased.

RECTANGLE is a list of quantities defining the associated rectangle: (RATIO LOC SIZE), height to width ratio, relative location, and size. The latter two quantities are not essential but save computation time, defining the relative location and size of the associated rectangle with respect to X. When a characterizer is first formed, its relative location and size are the same as that of its associated rectangle. However, at some later time, a new descriptor may be added to its description-set, enlarging the associated rectangle. Rather than recomputing all relative locations and sizes within the description-set and implication-set to renormalize the characterizer so that it has relative location (0, 0) and relative size 1.0, the associated rectangle is given an offset to accomplish the same purpose.

THRESHOLD has already been mentioned. It is a non-negative integer that is determined in the learning process as a function of the description weights in the description-set. When the cumulative description weight of an instance exceeds THRESHOLD, that instance is found.

TOTAL-DWT is the sum of the description weights in the description-set of X. It is used in the recognition process in determining how much effort has been expended on an instance, compared to how much progress has been made in finding it.

USE is the number of times characterizer X has been found in prior inputs, used in converting implication weights to probabilities.

BIRTHDATE is the frame number when X was first formed. It is used in determining the age of a compound characterizer. In the learning process, compounds are analyzed and removed from ITM after a certain age, to be discarded or added to LTM.

The COST of a characterizer influences when it will be investigated during the recognition process. The COST of a primitive is a number representing its relative computation cost and is determined by the programmer when the primitive is written. The COST of a non-primitive is the number of descriptors in its description-set times a constant, a rough measure of its processing cost during recognition.

VALUE also influences when characterizer X will be investigated. It measures the worth of X in characterizing its implicands, and ultimately in characterizing goals. The VALUE of a goal is specified by the trainer for each external pattern name, or arbitrarily set to 1. The VALUE of a non-goal is a function of the VALUEs of its implicands, their implication weights, and the description weights of X in the description-sets of its implicands.

(The actual function is the maximum over all implicands I_i of

$$(IWT(I_i) / USE(I_i)) \cdot VALUE(I_i) \cdot (DWT(X) / \text{maximum } DWT).$$)

1.5 Differences in the Representation of Primitives, Compounds, and Goals

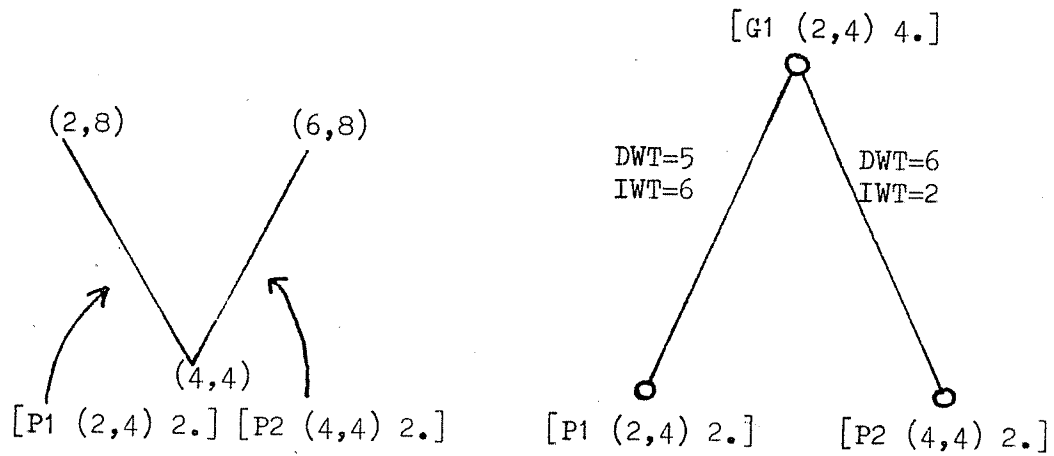
There are only minor structural differences between primitives, compounds and goals, for we have attempted to keep memory as homogeneous as possible. Primitives have a null description-set, since they are not defined in terms of other characterizers. Goals have a PRINTNAME and other characterizers do not.

The recognition process treats the three characterizer types identically, except for the following: Primitives are computed, but non-primitives, instead, have their description-sets investigated. When goals are found, in contrast to non-goals, their names are output and feedback may be given.

The learning process handles implication-sets of all three characterizer types identically, but distinguishes between them when modifying description-sets.

1.6 Some Examples of Characterizers

Figure 4-2 shows some simple characterizers; goal G1 represents LETTER-V, which has only two descriptors, line primitives P1 and P2. (We shall explain line primitives at length in Chapter 7.) Description weights of P1 and P2 are 5 and 6, and



Implication-set of P1 = [(G1 (0,0) 2.) 6]

Implication-set of P2 = [(G1 (-1,0) 2.) 2]

Description-set of G1 = [(P1 (0,0) .5) 5], [(P2 (1,0) .5) 6]

THRESHOLD of G1 = 10

Figure 4-2. The representation of LETTER-V, described by primitives P1 and P2.

the THRESHOLD of G1 is 10; therefore, both primitives must be found in order that G1 be found.

In terms of the graph representation of memory, links between nodes have been shown as undirected, but it is more accurate to consider a link as two bidirectional links, an implicand link and a descriptor link. The previously mentioned redundancy between implication-set and description-set is reflected in the fact that there are no unidirectional links.

2. The Instance

The instance is the unit of information storage in STM. STM does not, in fact, exist as a separate list. It is simply what we choose to call the set of all instances created during recognition. The primary function of the instance is to record the ongoing process of recognition. Suppose, for example, a HOUSE consisting of a SQUARE and TRIANGLE occurs in the input. Because HPL is not a parallel program, HOUSE will not be recognized "all at once". A part of SQUARE may be recognized, then perhaps all of SQUARE. An instance of HOUSE may be formed at this time, but this instance is not yet found; it only represents some characterizer which _may_ be present in the input. The instance of HOUSE will record that SQUARE has been found. When TRIANGLE is found, that information will also be recorded within the instance of HOUSE, and then that instance will finally be found.

As stored in LTM, a characterizer describes only relative

location and size relationships. An instance, in contrast, always corresponds to the occurrence (possible or actual) of a characterizer with some particular absolute location and size in the input. Thus one LTM characterizer may correspond to several instances. Suppose a word such as EYE has been presented. Only one LTM characterizer will represent the letter E, but there will be two instances representing each occurrence of E in the pattern.

The instance is similar in structure to the LTM characterizer, but contains additional information in order to record the recognition process. It consists of a description-set, implication-set, and miscellaneous data.

2.1 The Description-Set cf an Instance

The description-set of an instance is a list of descriptors. (Our terminology is the same as that used for the LTM characterizer. When confusion might arise, we shall explicitly say "descriptor of an instance", for example.) A descriptor contains, in addition to the same information as a LTM descriptor, a link to another instance. The form of a descriptor is

[((NAME LOC SIZE) DWT) MATCH].

The ((NAME...) DWT) part is identical in meaning to the LTM descriptor. (In fact it is identical in structure. In our LISP implementation, the LTM descriptor is not recopied in the instance; rather, there is a direct memory reference to the corresponding LTM descriptor. This reduces the size of an instance.)

MATCH is a link from the descriptor to an instance of NAME. Since a descriptor specifies relative location and size and an instance specifies absolute location and size, the MATCH link establishes a correspondence between relative and absolute quantities. Forming this link is called matching an instance with a descriptor. Before we give a simple example using the MATCH field, we shall describe the implication-set, which has a similar field.

2.2 The Implication-Set of an Instance

The implication-set of an instance is a list of implicands, each of which has the form

[((NAME LOC SIZE) IWT) IMATCH].

((NAME...) IWT) is identical to the implicand of a LTM characterizer (and again is represented by a memory reference to the actual LTM implicand).

IMATCH is a link from the implicand to an instance of that implicand. Our earlier comment about the theoretical redundancy of the implication-set applies to the IMATCH link as well, but using IMATCH is more efficient than recovering the link from the corresponding description-set MATCH field.

Both MATCH and IMATCH fields allow for only one instance to match.

2.3 An Example: Matching Instances to Descriptors and Implicands

An instance corresponds to one characterizer with a particular absolute location and size. We shall frequently use the characterizer reference [NAME LOC SIZE], where LOC and SIZE are absolute, in referring to an instance, keeping in mind, however, its underlying structure.

The MATCH and IMATCH fields play an important role in the recognition process; for example, an empty MATCH field may indicate that a new instance should be formed. This role will be described in the next chapter.

Figure 4-3 illustrates the matching process. The input is the pattern shown in Figure 4-2, LETTER-V (goal G1), having two primitives P1 and P2 as descriptors. Suppose instance [P1 (2,4) 2.] has already been found. [G1 (0,0) 2.] is an implicand of P1; hence, there is a chance that an instance of G1 will be created, namely [G1 (2,4) 4.] (the absolute location (2,4), for example, is computed from the absolute location of P1 and the relative location of C1 using the formula given in Section 1.12). In Figure 4-3(a), a MATCH link is formed from the description-set of instance [G1 (2,4) 4.] to the instance [P1 (2,4) 2.], as well as a corresponding IMATCH link from the implication-set of P1 to the instance of C1. Once the instance [G1 (2,4) 4.] has been formed, its absolute location and size are fixed, and hence the locations and sizes of all its descriptors are fixed. Instance [G1 (2,4) 4.] has an unmatched descriptor, [P2 (1,0) .5]. (The

```
Instance [P1 (2,4) 2.] with
    Implication-set = [((G1 (0,0) 2.) 6)• ]
Instance [G1 (2,4) 4.] with
    Description-set = [((P1 (0,0) .5) 5)•], [((P2 (1,0) .5) 6) *]
```

(a) Matching between an instance and descriptor of P1

```
Instance [P2 (4,4) 2.] with
    Implication-set = [((G1 (-1,0) 2.) 2) • ]
Instance [G1 (2,4) 4.] with
    Description-set = [((P1 (0,0) .5) 5) •], [((P2 (1,0) .5) 6)• ]
```

(b) Matching between an instance and descriptor of P2. The descriptor of P1 is assumed matched from (a).

Figure 4-3. Matching instances against descriptors and implicands (see text).

null MATCH field is indicated in the figure by an asterisk.) When [G1 (2,4) 4] is investigated, that unmatched descriptor will be detected and an instance [P2 (4,4) 2] will be formed. Figure 3-3(b) shows the new MATCH and IMATCH links between P2 and C1.

We shall describe <u>when</u> MATCH and IMATCH links are formed in the next chapter.

2.4 Tolerance: Not Requiring an Exact Match

We believe that flexibility, allowing variability and inexactness in pattern matching, is important. (See Uhr [1973] for several approaches to flexibility.) HPL embodies flexibility in two respects. The first has already been mentioned, the description weight/threshold structure. Depending on the relation between the threshold and the description weights, one, all, or various combinations of descriptors may "fire" a characterizer.

A second way in which HPL exhibits flexibility is in the TOLERANCE mechanism. Although our examples have not shown this, HPL does not require an exact match between the location and size of a descriptor and the location and size of an instance. They need only be closer than a certain fixed percentage, called TOLERANCE, for a match to succeed. For example, if a descriptor specifies a location of (4,2) and a corresponding instance is found at (4.2, 2.1), the match will still succeed.

2.5 Additional Items Stored in an Instance

Several items stored in an instance are part of neither the description-set nor the implication-set. Most of them are used in the recognition process; they will be explained in detail in the next chapter.

The characterizer reference [NAME LOC SIZE] specifies the characterizer name, absolute location and absolute size of the instance.

STATUS indicates the membership of an instance in one of three disjoint lists, ACTIVE, WAITING and COMPLETED, used during recognition.

LEVEL is a flag which is true if I is found but has no implicands that are found; that is, I is at the top level of the current graph of found characterizers. LEVEL is used in the learning process when forming new compounds and links.

Several interacting "weights" or tallies called <u>instance weights</u>, stored within an instance, are continually updated during the recognition process. They are CUM-DWT (cumulative description weight), REM-DWT (remaining description weight), AWT (active weight), FOUNDWT (found weight), IMPWT (implied weight) and COMPWT (composite weight). These weights and the MATCH and IMATCH fields direct the recognition process, record its current state, and determine whether an instance is found.

CHAPTER 5

THE RECOGNITION PROCESS

1. Introduction

The main function of the recognition process is determining what characterizers, especially goals (patterns), are present in the input. In so doing, it attempts to search memory efficiently and minimize the cost of computing primitives.

Memory search proceeds both upward and downward in terms of the graph representation of memory introduced in Chapter 3. Downward search occurs when a non-primitive characterizer is *investigated*, expanding its description-set into instances. Upward search occurs when a characterizer is FOUND, causing the formation of instances of its implicands. Downward and upward search correspond to the two subprocesses of the recognition process, *descriptor-processing* and *implicand-processing*.

First, we shall define some important terms, then present an overview and an example of recognition. We next give a detailed description of implicand-processing and descriptor-processing, and finally discuss some related issues.

1.1 Definition of Terms

The purpose of the recognition process is determining what instances of characterizers are FOUND in the input, in particular, what goals are FOUND. HPL also determines when instances are

definitely not present or NOT-FOUND, for it can then stop looking for them. Note that if an instance is not yet FOUND, it is not necessarily NOT-FOUND, for HPL simply may not have collected enough information to make a decision. An instance whose STATUS has been determined as either FOUND or NOT-FOUND is called a COMPLETED instance.

An instance of a primitive is FOUND when the feature or property it represents is present in the input. An instance of a non-primitive is FOUND when its cumulative description weight (CUM-DWT) exceeds its THRESHOLD. CUM-DWT, in turn, is the sum of the DWTs of its descriptors which have been FOUND.

An instance of a primitive is NOT-FOUND when the feature it represents is not present in the input. A non-primitive instance is NOT-FOUND when its remaining description weight (REM-DWT) is less than the total additional description weight needed to exceed THRESHOLD (i.e. THRESHOLD - CUM-DWT). REM-DWT is the sum of the DWTs of those descriptors which have not been COMPLETED. A NOT-FOUND instance cannot be FOUND because it cannot possibly accumulate enough additional description weight to exceed THRESHOLD. Suppose all descriptors of instance I have been COMPLETED except one, which has DWT = 5, hence REM-DWT = 5. If CUM-DWT is 15 units below THRESHOLD, there is no possibility that instance I can be FOUND, for even if the remaining descriptor is later FOUND, CUM-DWT is still 10 units below THRESHOLD.

2. An Overview of the Recognition Process

In the recognition process, two interacting subprocesses, descriptor-processing and implicand-processing, are alternately performed. The subprocesses operate on the ACTIVE list and the COMPLETED list, respectively, two lists which are accumulated during recognition. Both subprocesses can add instances to either list.

ACTIVE is a list of all instances which have been created but have not yet been descriptor-processed, i.e., "investigated". Whenever a new instance is formed, it is put on ACTIVE. The order of ACTIVE is determined by the active weight (AWT) of each instance composing it. The active weight of an instance is continually updated as information about its descriptors or implicands accumulates in processing the input.

COMPLETED is a list of all instances which are either FOUND or NOT-FOUND, but have not yet been implicand-processed. It is ordered with respect to the VALUE of its instances.

ACTIVE and COMPLETED direct the recognition process. The AWT of the top instance on ACTIVE is compared with the VALUE of the top instance on COMPLETED. If the AWT is sufficiently large (see Section 5), descriptor-processing will be entered; otherwise, implicand-processing will be entered.

A third list, WAITING, is a default list of instances. After an instance has been descriptor-processed and is removed from ACTIVE, but before it is COMPLETED, it is WAITING for descriptors

to be processed.

Descriptor-processing investigates instances, computing primitives or forming instances of descriptors of non-primitives. Computed primitives are added to COMPLETED; newly formed instances of descriptors are added to ACTIVE. For non-primitives, descriptor-processing is analogous to creating subgoals in order to achieve a higher-level goal.

Implicand-processing, which operates on the COMPLETED list, has two main concerns: updating the <u>instance weights</u> associated with instances of implicands (described in Section 4) and forming new instances of implicands where they do not already exist. After the instance weights of an instance are updated, that instance may, in turn, be COMPLETED.

Naming of patterns is a byproduct of the two subprocesses. When a goal is both COMPLETED and FOUND, its name is printed and feedback may be given.

In general, the subprocesses of the recognition process search memory, forming instances of characterizers as they proceed. New instances go to the ACTIVE list, COMPLETED instances to the COMPLETED list. Implicand-processing starts from a given COMPLETED instance and forms instances of its implicands, an upward-moving process. There is no further upward search from an implicand until it is also COMPLETED. HPL attempts to find descriptors of an unCOMPLETED instance, by forming instances of those

descriptors. This is a downward-moving process, accomplished by descriptor-processing. Eventually memory search will extend down to the primitives, which can be computed and thus COMPLETED. The information that a primitive is COMPLETED is sent up to its implicands via the matching of MATCH and IMATCH fields and instance weights. If an implicand is in turn COMPLETED, then that information is fed upward to its implicands, and so forth.

3. An Example of the Recognition Process

Let us assume that goal HOUSE contains SQUARE and TRIANGLE as descriptors and that an instance of SQUARE has just been FOUND. That instance will be added to the COMPLETED list. If the VALUE of SQUARE is high enough, it will be the next instance implicand-processed. Then, instances of implicands of SQUARE, such as HOUSE, BLOCK, and FACE, will be formed and placed on the ACTIVE list with AWTs depending on their VALUEs, likelihoods, and COSTs. Those instances will have MATCH fields linked to the instance of SQUARE which caused their formation. (Similarly, SQUARE will contain IMATCH fields linked to the new implicand instances.) If none of the AWTs are high enough, the next instance on COMPLETED will be implicand-processed. Suppose, however, that HOUSE has a high AWT and that HPL switches to descriptor-processing of it.

HOUSE will be investigated. Since it is not a primitive, its descriptors will be scanned. Its descriptor SQUARE is already

matched to a FOUND instance (in its MATCH field), but its descriptor TRIANGLE is unmatched. A new instance of TRIANGLE will be created and added to ACTIVE (and matched in the MATCH field of SQUARE, etc.); HOUSE will be removed from the ACTIVE list and added to the WAITING list. If its AWT is sufficiently high, then TRIANGLE will be investigated, and instances of the straight line primitives describing it will be formed, and so forth. Eventually, the primitives may be FOUND. The instance weights of their implicands, including TRIANGLE, will be updated, and TRIANGLE will be FOUND. The instance weights of its implicands, including HOUSE, will be updated, and HOUSE will be FOUND.

4. Weights Associated with an Instance

Several weights associated with each instance, collectively called *instance weights*, are continually updated during the recognition process. They determine the active weight of the instance and whether it is FOUND or NOT-FOUND. Exactly when these weights are updated is explained in Sections 5 and 6. In general, however, they are updated when new information is discovered. For example, when an instance is COMPLETED, the instance weights of its implicands are updated, which may in turn cause them to be COMPLETED.

We shall use the following notation: X is the instance whose weights will be described. D' is a descriptor of X, and D is an instance matched to D' (in the MATCH field). Similarly, I' is an implicand of X, and I is an instance matched to I' (in the IMATCH

field). It is important to remember that if D' is a descriptor of X then X is an implicand of D'. Thus we may talk about the description weight of D' in the description-set of X as well as the implication weight of X in the implication-set of D'.

As indicated in Section 2, the active weight (AWT) of an instance is very important in recognition, for it determines the order of instances on ACTIVE, which, in turn, determines the order in which instances are descriptor-processed. Similarly, the VALUE of an instance determines the order of instances on COMPLETED, thus the order in which implicands are implicand-processed.

VALUE is a parameter associated with a characterizer name in permanent memory, rather than an instance weight, for it does not vary during recognition. It is a measure of the "worth" of a characterizer in describing its implicands, and, ultimately, in describing goals. The VALUE of a goal is determined by the trainer, or else is set to 1. The VALUE of a non-goal is a function (precisely defined in Chapter 4) of the VALUEs and IWTs of its implicands. It is thus a reasonable quantity to use in determining when an instance should be implicand-processed.

AWT depends on the likelihood, VALUE, and COST of an instance and is therefore a reasonable quantity to use in determining when an instance should be descriptor-processed. It is the product of COMPWT, VALUE and (1 - COST • constant). COMPWT, a measure of the likelihood of an instance, is explained below. COST, defined in Chapter 4, is a measure of the processing cost of an instance.

The constant is a normalizing factor which determines the degree to which COST affects AWT.

CUM-DWT and REM-DWT have already been discussed. CUM-DWT (cumulative description weight) is incremented by the description weight of D' when instance D, matched to D', is FOUND. REM-DWT (remaining description weight) is decremented by the DWT of D' when matching instance D is COMPLETED, whether it is FOUND or not.

FOUNDWT and IMPWT are used in computing COMPWT. FOUNDWT (found weight), the square of CUM-DWT divided by THRESHOLD, is a measure of how close CUM-DWT is to THRESHOLD. If X is FOUND, then FOUNDWT is 1.0. (An alternative is the "partially found" approach discussed in Chapter 8.)

IMPWT (implied weight), a measure of the likelihood X will be FOUND, is composed of two factors, one depending on its FOUND descriptors and the other on its WAITING implicands.

The first factor in IMPWT is obtained by adding* to IMPWT the IWT (divided by USE to make it a probability) of X in the implication-set of every FOUND instance D of a descriptor of X. This is a measure of the likelihood of X given the occurrence of its FOUND descriptors.

The second factor is obtained by adding to IMPWT the product of the DWT of X in the description-set of each of its WAITING

---

*Rather than simple addition, the formula for combining independent probabilities, $a + b - a \cdot b$, is used, even though probabilities are not guaranteed to be independent.

implicands, a normalizing constant, and the COMPWT of those implicands. It reflects the subgoal/goal relationship between X and its WAITING implicands. Because of this factor, instances of descriptors of WAITING instances will have greater IMPWTs, hence greater AWTs, and consequently greater chance of being investigated. If an instance is COMPLETED but NOT-FOUND, this factor causes IMPWTs to be decremented rather than incremented, decreasing the chance the descriptors will be investigated. (IMPWT is not simply set to zero, for an instance may be a descriptor of more than one characterizer.)

COMPWT (composite weight) is a function of FOUNDWT and IMPWT, lying between them in value. The greater the value of TOT-DWT - REM-DWT, a measure of the effort exerted on this instance, the closer COMPWT is to FOUNDWT. The lower it is, the closer COMPWT is to IMPWT. In other words, the greater the effort expended on an instance, the more COMPWT reflects FOUNDWT, i.e., what is known about the occurrence of this instance, as determined by its FOUND descriptors. The less effort expended, the more COMPWT reflects IMPWT, or what other instances estimate regarding the presence of the instance.

5. Implicand-Processing

Implicand-processing updates instance weights of implicands of COMPLETED instances, and forms new instances of such implicands where appropriate. Figure 5-1 shows a general flowchart of

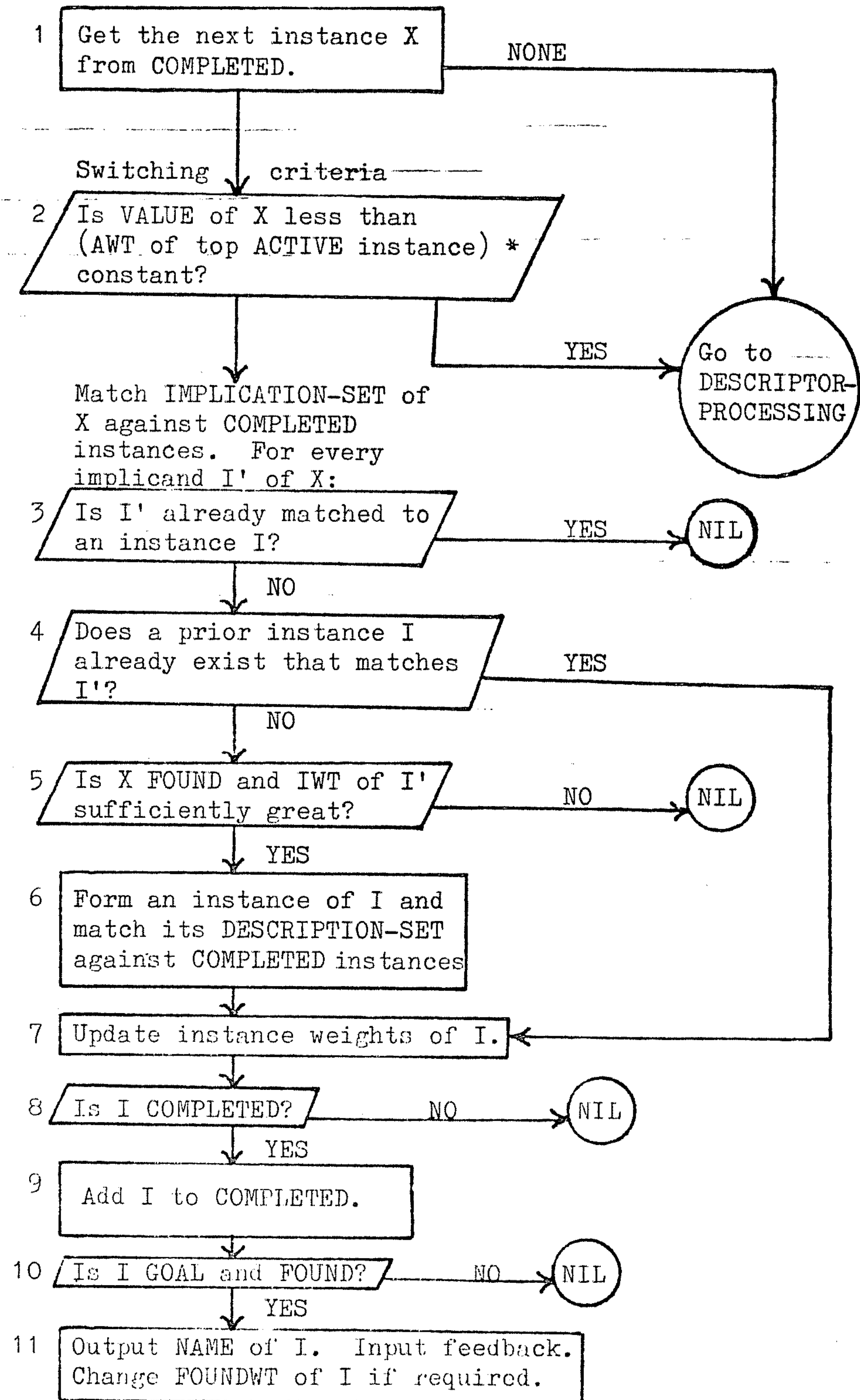


Figure 5-1. A general flowchart of implicand-processing.

implicand-processing. The numbers on the flowchart correspond to numbers in the following discussion. In Section 7 we discuss alternative methods to begin the recognition process. But for now, let us assume some instances are already on the COMPLETED list.

Implicand-processing operates on the COMPLETED list, processing each COMPLETED instance until the VALUE of the next instance is less than the AWT (times a fixed contant) of the top ACTIVE instance. Then HPL switches to descriptor-processing of ACTIVE. The constant is a parameter which controls the degree of switching between implicand-processing and descriptor-processing. If it is 0, for example, implicand-processing occurs as long as there is anything on COMPLETED. (HPL appeared to work best with the constant set to 1.0.) [Boxes 1 and 2]

After an instance is implicand-processed, it is removed from COMPLETED; an instance is implicand-processed only once. Similarly, an instance is placed on ACTIVE to be descriptor-processed only once.

The main task of implicand-processing is forming new instances of implicands. When this occurs, memory search is extended upward. Suppose HPL is processing implicand I' of instance X. If I' is already matched to instance I, then nothing more is done with the implicand. This means I has previously been descriptor-processed at which time the match was made [Box 3]. If there is no match, HPL determines if there is already an instance of the same NAME, absolute location, and size (within TOLERANCE) as I'. If so, then

IMATCH of I' is set to I. Similarly, the corresponding MATCH field in the description-set of I is set to X. (HPL efficiently handles the search for the appropriate instance. Rather than searching through all instances, the much shorter list of INSTANCES associated with the NAME of I' is searched.) [Box 4]

If no prior instance is located, an instance of I' may be created and added to ACTIVE. Such an instance, however, is only formed under certain circumstances. First of all, instance X must be FOUND. This is an important restriction, for it means that HPL only investigates the implications of positive occurrences of characterizers. If an instance is NOT-FOUND, no implications of that fact are pursued. (HPL does, however, lower IMPWTs and AWTs of descriptors of a NOT-FOUND instance, as already mentioned.) Secondly, only sufficiently likely implicands are pursued. The IWT of I' divided by USE must be greater than a fixed parameter [Box 5].

Determining when instances are FOUND requires updating instance weights. Instance weights are updated on two occasions: when an unmatched descriptor matches an instance previously COMPLETED and when an instance already matching a descriptor has just been COMPLETED.

Whenever a new instance is formed, its description-set is scanned to determine what descriptors have already been COMPLETED [Box 6].

Instance weights of I are updated to take into account that X is COMPLETED [Box 7]. A test is made to determine if I is

COMPLETED, and if so, it is added to the COMPLETED list [Boxes 8 and 9].

If I is a goal and has been FOUND, its NAME is output. Feedback may be given at this time. If feedback indicates the response is wrong, the FOUNDWT of I will be set to 0, and it will be regarded as NOT-FOUND [Boxes 10 and 11]. Implicand-processing then continues with the next instance on the COMPLETED list until HPL switches to descriptor-processing.

The following mechanism is not shown in Figure 5-1. Whenever an instance is COMPLETED (whether within implicand-processing or descriptor-processing), its implicands which are <u>already</u> matched have their instance weights immediately updated and are tested to determine if they too are COMPLETED. Implicand-processing, which may be deferred because of the VALUE vs. AWT test [Box 2], only affects implicands not already matched. Implicands already matched are WAITING for completion of descriptors, and when a descriptor is COMPLETED, the WAITING instance is immediately notified. This mechanism prevents the updating of WAITING instances from being postponed indefinitely.

6. Descriptor-Processing

Descriptor-processing investigates instances by computing primitives or expanding non-primitives. Figure 5-2 shows a general flowchart of descriptor-processing.

First the termination criteria are tested (see Section 8).

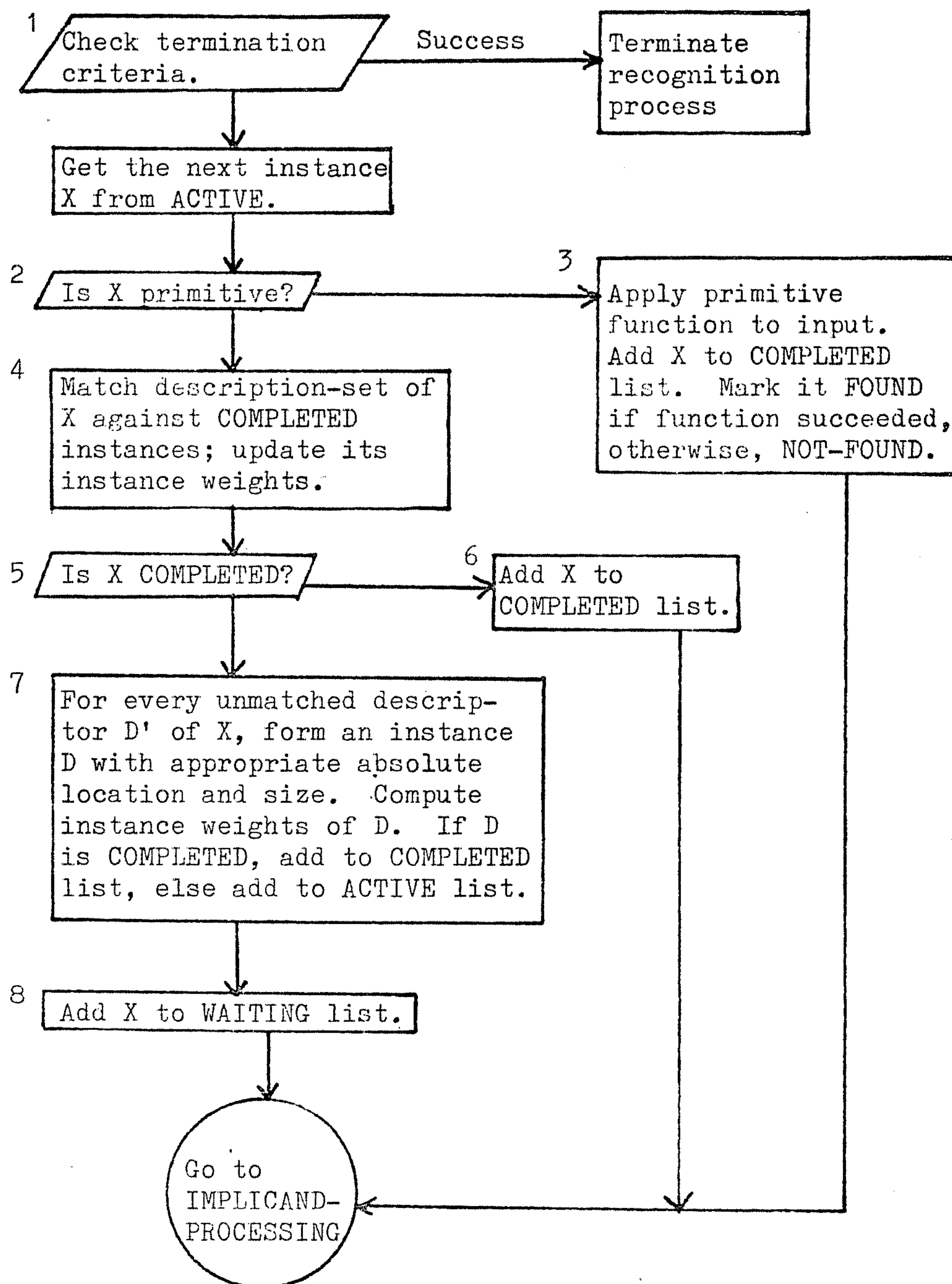


Figure 5-2. A general flowchart of descriptor-processing.

If they succeed, the recognition process ends [Box 1].

If the top instance X of ACTIVE, which has the highest AWT, is a primitive, it is computed. A primitive function succeeds or fails, depending on whether the property is found with the specified absolute location and size (within TOLERANCE). Success means it is FOUND, failure NOT-FOUND. In either case, the primitive is added to COMPLETED [Boxes 2 and 3].

If X is not a primitive, its description-set is matched against the list of COMPLETED instances to see if any descriptors are already COMPLETED, and its instance weights are updated [Box 4]. If X is now COMPLETED, no further processing of its descriptors is necessary, and it is added to COMPLETED [Boxes 5 and 6].

Otherwise, for every unmatched descriptor D' of X a new instance D is formed with appropriate absolute location and size. This extends memory search downward. As with any new instance, D' is matched against COMPLETED, and its instance weights are updated. If D is COMPLETED, it is added to the COMPLETED list; otherwise, it is added to ACTIVE [Box 7].

X becomes a WAITING instance and will remain so unless it is eventually COMPLETED [Box 8}.

7. Starting the Recognition Process

We have described how descriptor-processing operates on ACTIVE and implicand-processing operates on COMPLETED. To start the recognition process, one or more instances must initially be

placed on one of these two lists. Several alternatives will be described for so doing.

(1) A fixed list of characterizers could be placed on ACTIVE at the beginning of each frame, building-in what HPL should look for first. Initial tests of HPL used this method. (2) A list of characterizers could be suggested by the trainer at the time the pattern is input. We have not used this alternative. (3) We shall call the following alternative "preprocessing". A fixed set of primitives can be applied in "parallel" (i.e., without regard to order) to the input before recognition begins. The result of this computation is immediately placed on COMPLETED. This technique was devised for use with LINE primitives, where it seemed particularly appropriate (see Chapter 7). With this approach, the recognition process cannot compute primitives efficiently, since they are all computed initially, but it will still search memory efficiently. (4) The most attractive alternative is to compute, before a pattern is presented, an a priori AWT for each primitive in memory, to order them on the initial ACTIVE list. This AWT would be computed using the algorithm described in Section 4, except that

USE / (current frame - BIRTHDATE),

the a priori probability of the primitive, would be used instead of COMPWT. This approach, which allows HPL to learn the best primitives to look for first, was used with submatrix primitives (see Chapter 7).

8. Terminating the Recognition Process

Recognition is a sequential process in which decisions are made about the order in which to investigate characterizers. For this to have meaning, it must be assumed that all characterizers will not eventually be investigated. The recognition process should be terminated before all characterizers are investigated. There are several alternative stopping criteria.

(1) The simplest alternative is to terminate recognition after recognizing a pattern. But this will not work since there may be several patterns in the input. (2) The recognition process could be terminated by the trainer after HPL has recognized all possible patterns and subpatterns. (3) The trainer in the previous alternative could be simulated, by giving a subroutine a list of all patterns and subpatterns in the input. (4) Recognition could automatically terminate after a certain amount of computer time has elapsed. (5) Termination could occur when some function of processing variables exceeds a fixed limit. A function of the following quantities might be used: the current cumulative COST of primitives that have been computed, the number of instances created, the number of loops through descriptor-processing and implicand-processing.

For non-interactive runs of HPL, a combination of alternatives (3) and (4) was used. In interactive runs, (2) was used.

9. Response and Feedback

Whenever HPL finds a goal, it outputs its name, location and

size; therefore, it tends to produce a detailed description of a pattern. For instance in the example of FACE in Figure 1-1(c), HPL will output the names of the subparts of FACE, such as NOSE and MOUTH, before it outputs FACE. An advantage of this approach is that immediate feedback may be given to an incorrect response, causing HPL to immediately correct its error. A misnamed subpattern may be corrected, helping HPL to correctly name the pattern containing the subpattern. (Because of the weight and threshold structure, of course, there is already some built-in tolerance to error.) A disadvantage, however, is that HPL may produce more detail about a pattern than is desired. We may only wish to know if FACE is present, not if NOSE is present.

One approach to this problem is for HPL to output only top-level goals it has FOUND (goals with no implicands also FOUND), rather than all goals. In other words, no subpatterns would be output. If HPL were asked for more detail, it would output the names of the next highest-level goals. However, this is not a real solution. HPL would still find the goals in the same order; their names would just not be output. This reflects a significant problem of most pattern recognizers, including HPL. It seems much more reasonable that HPL should recognize FACE _before_ it has definitely recognized NOSE, that the whole should usually come before its parts, or at least concurrently. Asking HPL for more detail should involve more recognition. We give our ideas on this problem in Chapter 8 in discussing partially found characterizers.

HPL's output does not adequately represent its ability, for HPL independently outputs each pattern name that it finds, but does not indicate the subpattern/pattern relationships that are actually stored in memory. If HPL recognizes a HOUSE consisting of a TRIANGLE and SQUARE, for example, HPL will output TRIANGLE, SQUARE and HOUSE, but will not indicate that TRIANGLE and SQUARE are <u>subpatterns</u> of HOUSE, although this information was, in fact, used in deciding that HOUSE was present. We did not include this information as part of HPL's output, since we were closely monitoring the memory structure, and hence were aware of the subpattern/pattern relationships. An improved version of HPL, however, should explicitly output these relationships, an easy modification to the output routine.

When HPL outputs a name, YES or NO feedback may be given. If the feedback is NO, the FOUNDWT of the goal instance is set to 0, and it is added to a list of goals incorrectly FOUND, used in the learning process. YES is equivalent to null feedback, since description weight adjustment of goals only occurs when a response is wrong.

After this feedback, or when the recognition process terminates, a list of [NAME LOC SIZE] elements may be input as additional feedback, indicating patterns that should have been FOUND but were not. If a NAME has never been presented before, HPL forms a new goal node to represent it. If such a goal already exists,

HPL determines if there is a prior instance of it. If not, a new instance is formed. In either case, the goal instance is added to COMPLETED as FOUND. It is also added to a list of goals which should have been FOUND but were not, used in the learning process.

CHAPTER 6

THE LEARNING PROCESS

1. Introduction

We believe HPL's memory structure is theoretically powerful (and even more so with extensions described in Chapters 8 and 9). The practical performance of HPL, however, depends on how much it can learn. Here we discuss our approach to learning.

The learning process uses the information accumulated during recognition within COMPLETED instances. It consists of several subprocesses: weight adjustment, addition and deletion of descriptors, and compound formation and evaluation. Weight adjustment is the most straightforward of the subprocesses, since it changes weights of links that already exist. The other subprocesses change the structure of memory. The subprocesses are tentative, first-step learning mechanisms. At this point, we feel that the enumeration of the subprocesses is as important as the techniques used to implement them.

Before discussing the subprocesses, we shall explain the significance of compounds. This explanation was deferred until the recognition process had been described.

2. The Significance of Compounds

HPL can recognize patterns without using compounds, since nothing in the recognition process requires them. The learning

process requires compounds only in those subprocesses that exist for their creation and evaluation. Nevertheless, compounds are important and will be explained in the next few sections.

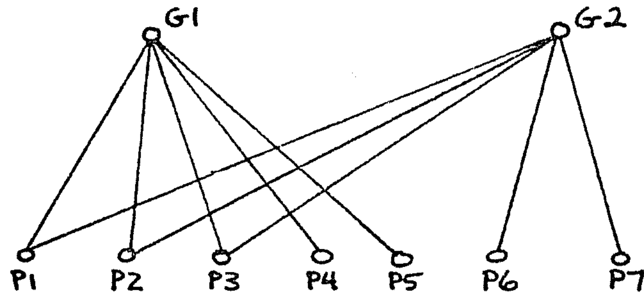
2.1 Conditional Probabilities and Compounds

Figure 6-1(a) shows two goals with primitives as descriptors. Suppose P1 has been FOUND. HPL must decide what primitive to compute next. If Pr(Pi/P1)* for $i \neq 1$ were available, HPL could choose the primitive Pi that maximized some function of Pr(Pi/P1), VALUE(Pi) and COST(Pi). Suppose P2 has been chosen and FOUND. At this point HPL cannot use Pr(Pi/P2) for its next decision but ideally requires Pr(Pi/P1 _and_ P2). This "ideal" approach requires that _all_ possible conditional probabilities be kept, and quickly blows up numerically. Let us proceed along a different route.

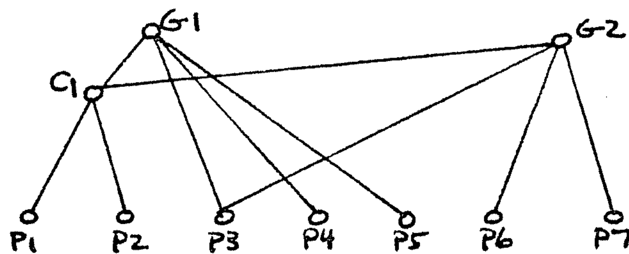
There is indirect information which might approximate Pr(P2/P1). Pr(G1/P1) is given by IWT(G1, P1) (literally the probability of finding some combination of primitives P1 through P5 sufficient to exceed THRESHOLD(G1), given P1). Consider DWT(P2, G1) as an approximation of Pr(P2/G1) (a loose interpretation of DWT. See Section 3.) It would be convenient if Pr(P2/P1) could be retrieved as the product of Pr(G1/P1) and Pr(P2/G1), that is, the product of IWT(G1, P1) and DWT(P2, G1),

---

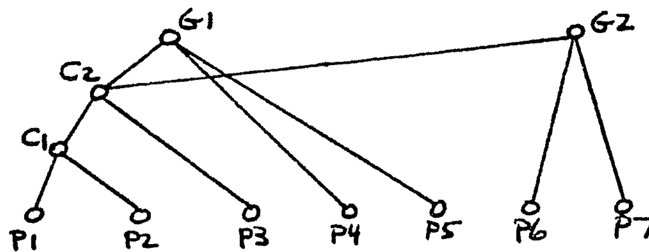
*Pr(X/Y) is the probability of finding characterizer X given that characterizer Y is already FOUND. Also, IWT(X,Y) is the IWT of implicand X in characterizer Y, DWT(X,Y) is the DWT of descriptor X in characterizer Y, VALUE(X) is the VALUE of characterizer X, etc.



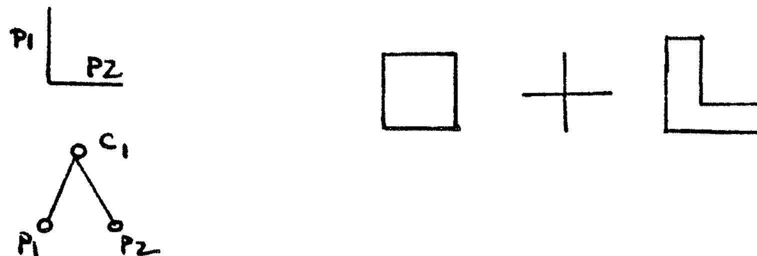
(a) Goals represented without compounds



(b) The same goals with compounds



(c) A compound containing a compound



(d) A specific compound and patterns containing it

Figure 6-1. Some simple examples of memory with and without compounds.

since this information is stored in memory. (HPL uses both IWT and DWT in computing the two factors of IMPWT, as explained in the previous chapter.) Unfortunately, this in general is not possible, for P1 and P2 may occur together in other goals besides G1, such as G2 shown. Computing Pr(P2/P1) would be mathematically complex, requiring conditional probabilities between goals. All potentially relevant probabilities would have to be stored. We would again have a situation that blows up numerically.

Figure 6-1(b) shows the preceding situation with the addition of compound C1 composed of P1 and P2. Pr(C1/P1) or IWT(C1, P1) is the likelihood of finding C1 given P1. Since C1 is P1 and P2, this probability is just Pr(P2/P1), the probability we were seeking. The existence of a compound thus allows the expression of conditional probabilities between its descriptors.

So far, however, we have only redefined the problem. To express all possible conditional relations would require creating compounds for every possible pair of characterizers. Suppose C1 contained P3 as an additional descriptor. Pr(C1/P1) would no longer exactly represent Pr(P2/P1), but it would still better approximate it than if C1 contained even more descriptors, or if C1 did not exist at all. The more descriptors in a compound, the less information it provides about individual conditional relationships. But the advantage of smaller compounds must be weighed against the disadvantage of producing too many compounds.

Compounds of compounds express some of the conditional information needed as recognition progresses. Figure 6-1(c) shows compound C2 with C1 and P3 as descriptors. IWT(C2, C1) is Pr(C2/C1), which is Pr(C2/P1 and P2).

Compounds capture conditional probabilities that facilitate recognition. Too many compounds, however, may have the opposite effect because of the additional processing time each instance requires. The question of exactly when compounds are formed is discussed in Section 5. HPL forms a small number of compounds containing several descriptors to try to express the most commonly occurring relationships.

2.2 Compounds and Allocation of Effort

Compounds, in the long run, should reduce the number of instances formed during recognition. Suppose P1 has been FOUND. Without compounds, as in Figure 6-1(a), instances of both G1 and G2 may be formed; HPL may consider both G1 and G2 as tentative input patterns. In Figure 6-1(b) if P1 is FOUND, only C1 will be considered; an instance of it will be formed. If C1 should be FOUND, *then* G1 and G2 will be considered. (In the short run, however, before HPL has decided which compounds are good and which should be discarded, more instances may be formed than are necessary.)

G1 in Figure 6-1(a) contains a larger number of descriptors

than C1 in Figure 6-1(b); consequently, G1's processing COST is greater than C1's. Investigating G1 involves a greater commitment of effort than investigating C1 (although the recognition process attempts to handle this commitment efficiently). Compounds break up larger units (goals) into smaller units, allowing HPL to move in smaller steps with more reliable information about its progress at each step (through success or failure of compounds).

Figure 6-1(d) shows a simple example of compound C1 composed of vertical line P1 and horizontal line P2 (ignoring location and size for simplicity), and several named patterns which contain the compound. Suppose P1 has been FOUND and that it is a part of a great many different patterns in addition to the ones shown containing C1. It is advantageous to determine whether P2 is present, for if it is, it narrows the range of possible patterns. The existence of C1 makes it easier for HPL to determine that P2 is worth early investigation.

2.3 Compounds May Be Good Characterizers

Compounds may make "better" characterizers than the individual descriptors composing them; that is, some combination of features may be better than the individual features themselves. In Figure 6-1(d), knowing that a vertical line (P1) or a horizontal line (P2) occurred may be much less important than knowing

that they both occurred at the same time and place in the input (C1). Obviously, if HPL is to evaluate compounds according to their worth in describing patterns, something more quantitative than this is necessary. The VALUE of a characterizer is our measure of its worth. A characterizer's VALUE (precisely defined in Chapter 4) is a function of its DWT in characterizers where it is a descriptor and the likelihood of those characterizers as given by IWT. HPL tries to create compounds that will have high VALUE.

2.4 Compounds Provide Flexibility

We have already discussed how weighted links and thresholds add flexibility in determining the presence of a characterizer. Compounds allow for many flexible nodes in describing a goal, rather than just the single node of the goal itself, for each compound contains weighted links and a threshold.

2.5 Compounds versus Goals

In Figure 6-1(d), compound C1 might be a representation of the letter L. Why is C1 a compound and not a goal? Compounds are formed for reasons "internal" to HPL, namely, the mechanism of compound formation. A compound may happen to correspond to something we choose to call a pattern. A goal characterizer would represent that pattern. There is nothing to prevent that compound from being a descriptor of the goal; it might even be the only descriptor. Thus there is no loss of generality in not assigning

pattern names to compounds. To reiterate: compounds are formed for internal reasons, goals for external reasons.

3. Inductive Weight Adjustment

Weight adjustment is the most straightforward subprocess of the learning process.

Implication weights of implicands of instance X represent the likelihood of finding implicands, given that X has been FOUND. Suppose instance X is FOUND. Every implicand of X which has a match (in the IMATCH field) to a FOUND instance has its IWT incremented by 1; otherwise, there is no change in IWT. By updating USE of X by 1 whenever X if FOUND, IWT / USE gives the desired probability.

For goals, adjustment of description weights is similar to that used in many learning pattern recognition programs. DWTs are adjusted only when an error is made: either HPL has made an incorrect response or has not named a goal which should have been named. In the first case, DWTs of descriptors matching FOUND instances (in the MATCH field) are decremented by 1 and those matching instances not FOUND are incremented by 1. This negatively reinforces the links that caused the incorrect response. In the second case, DWTs of FOUND descriptors are incremented and other descriptors decremented. This positively reinforces the links that produced the desired response. There are many possible variations of this technique [See Uhr, 1973]. One variation which might increase the learning rate would vary the weight

adjustment increment by the magnitude of the error. A measure of the error would be given by the difference between CUM-DWT of the instance and THRESHOLD.

Compounds are more problematic than goals. External feedback indicates if a goal is actually FOUND, whether or not its description-set has exceeded THRESHOLD. There is no such external evidence for compounds, and so it is illogical to say that a compound which has been FOUND is wrong, that is, has not really been FOUND. Our learning technique reweights FOUND compounds. For any FOUND compound, if a descriptor is FOUND its DWT is incremented by 1 (up to some maximum); otherwise, it is decremented by 1. The THRESHOLD of the compound is also recomputed to reflect the new total description weight; it is set at some fixed fraction of TOT-DWT, in most of our tests $\frac{3}{4}$ of TOT-DWT. Our intention is to perturb the description of the compound slightly in the hope of increasing its VALUE.

Because reweighting is slightly different for compounds and goals, the meaning of the description weight is somewhat different. For compounds, the DWT reflects the relative frequency of occurrence of descriptors. For goals, where reweighting only occurs on an incorrect response, it is more accurate to say that DWT reflects the "importance" of the descriptor in describing the goal. We would prefer a more uniform treatment of compounds and goals.

4. Adding and Deleting Links

In this section we discuss mechanisms to structurally modify characterizers by adding and deleting links (descriptors and implicands). Because of the relationship between descriptors and implicands previously explained, when descriptor D is deleted from (or added to) the description-set of characterizer X, the implicand corresponding to X in the implication-set of D is also deleted (or added). The purpose of adding and deleting links is to make small perturbations in the memory structure in order to search for better characterizers.

The mechanism for deleting links is quite simple. When a description weight reaches zero, the descriptor is deleted from the characterizer. (Even here, there are other alternatives. DWTs could be allowed to decrease below zero. Finding a descriptor with a negative DWT would suggest the non-presence of the characterizer it is part of; however, it would still be desirable to remove descriptors whose DWT is zero over the long run. We further discuss negative DWTs in Chapter 9.)

There are two cases where links may be added to a characterizer. When a new compound is formed, implication links will be added to those characterizers in its description-set. This is a natural consequence of compound formation and is discussed in the next section. The other case occurs when a new link is formed between a characterizer and a goal; that is, a new descriptor is added to the description-set of a goal. Although several

goals may be present in a scene, for simplicity new links are formed with only one particular goal during a frame, either the last name output by HPL or the last name submitted as feedback. This name is assumed to be the overall name of the scene. With a single pattern, this is just the pattern name. If there is no overall name, then this process will not occur. Any top level characterizer (a characterizer that is not a descriptor of any other characterizer also FOUND) which has been FOUND and is not already a descriptor of the goal will be made a descriptor. This technique may add irrelevant descriptors; we argue that weight adjustment will take care of such characterizers by eventually downweighting them to zero. When a new descriptor is added to a goal, its THRESHOLD is incremented by the initial description weight, a fixed constant.

5. Compounds: Their Formation and Evaluation

Ideally, HPL should profligately form compounds from nearly any instance in the COMPLETED list, for it can never be known in advance which are the "valuable" compounds. Weight adjustment and compound evaluation would eventually weed out the many bad compounds. Practically, this approach would be excessively wasteful of computer time. We selected some criteria that would reduce the number of compounds but not be overly restrictive.

A new compound is formed from instances that are FOUND, top level, not members of ITM, and sufficiently close together. Any

potential compound meeting those criteria will be created. A new compound must have more than one but less than six descriptors. The size limit and criterion for "sufficiently close together" are fixed parameters.

Requiring that instances be close together might seem unduly restrictive. The subroutine which determines if two instances are close enough together compares their associated rectangles to see if they touch at any point, plus or minus a fixed tolerance. In general, higher-level characterizers have larger associated rectangles than lower-level ones, since their associated rectangles include the associated rectangles of their descriptors. Therefore as higher-level compounds are formed, they cover more and more area. In other words, a compound is a _local_ collection of co-occurring characterizers, but local relative to its associated rectangle.

When a compound is created, it is added to ITM, the temporary storage area for compounds, and the appropriate descriptor and implicand links are formed. The THRESHOLD of the new compound is initially set at $\frac{3}{4}$ of the TOT-DWT of its description-set.

At a fixed age, a compound is evaluated. After evaluation, it is removed from ITM and either discarded or added to LTM. ITM and compound evaluation constitute a filtering process that weeds out the possibly large number of bad compounds. Once a compound is

added to LTM, it remains there permanently. (If HPL were given many different patterns so that LTM became very large, it might also be necessary to monitor LTM, discarding rarely used characterizers.)

Several quantities might be used to evaluate compounds, but they are not all independent. For example, VALUE, or VALUE, COST, and probability, or USE vs. AGE, might be used. USE, the number of times the compound has been FOUND, depends not only on whether the compound could theoretically be FOUND, but if it was actually accessed from memory, investigated, and FOUND. This, in turn, depends on the factors that enter into the AWT computation, its VALUE, COST, and so forth. We chose to discard a compound if its VALUE did not exceed the maximum VALUE of its descriptors.

CHAPTER 7

EXAMPLES OF PATTERN RECOGNITION BY HPL

1. The Choice of Primitives

In this chapter, we shall demonstrate the present recognition ability of HPL, giving as examples patterns recognized in computer runs. More advanced recognition ability with modifications to HPL is discussed in Chapter 8.

HPL accepts primitives of the form: [NAME LOC SIZE]. The primitives compute functions either FOUND or NOT-FOUND in the input.

Three types of primitives have been used--letters, submatrices, and lines. Most runs have used the latter to reduce computing costs. In this section, we discuss the three types of primitives. Several pattern recognition problems handled by HPL are explained in later sections.

1.1 Letter Primitives

For initial debugging of HPL, primitives of the form [LETTER-A LOC SIZE], [LETTER-B...], etc., were used. If HPL requested the computing of primitive LETTER-A at (4,0), an "A" would be looked for in the input at (4,0) as a primitive (i.e., not in terms of its components). Inputs were one-dimensional strings of letters up to some fixed maximum length. All letters

were the same size, and so the SIZE parameter was in effect ignored. A typical memory structure built up from letter primitives is shown in Figure 7-1(a).

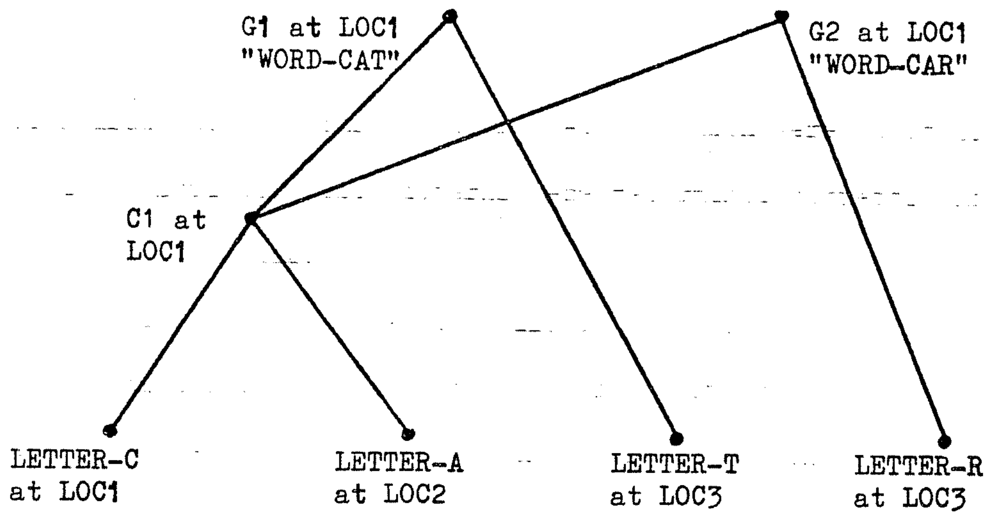
More realistically, letters are not primitives but higher-level characterizers composed of lower-level primitives. Using lower-level primitives, the same structure as the one shown in Figure 7-1(a) could be built up, except that the graph would extend below the nodes representing the letters, to describe their more detailed structure. For example, LETTER-A might be described in terms of more basic strokes, as in Figure 7-1(b). This suggests that the distinction between a primitive and a non-primitive is a decision as to the level at which to build in basic functions. HPL's structure treats primitives and non-primitives in a unified manner, except that primitives are computed and non-primitives are "investigated".

We shall not further discuss this simplest kind of primitive.

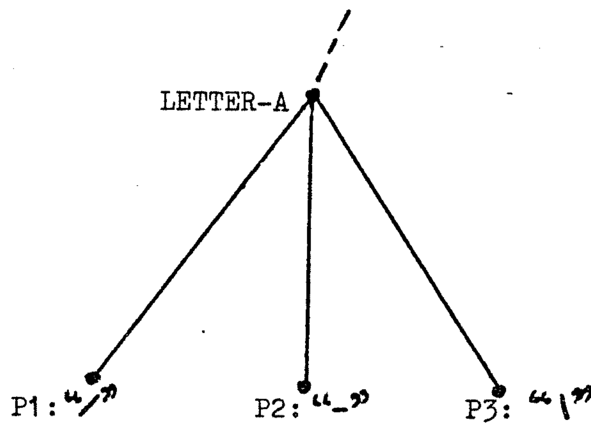
1.2 Submatrix Primitives

When using submatrix primitives, the input scene is in the form of an M x N matrix of 0 or 1 cells. A submatrix primitive is a fixed template specified as an m x n submatrix of 0 or 1 cells, where m and n are less than M and N. Figure 7-2(a) shows several submatrices. The submatrix is compared cell by cell with an m x n part of the input; if they are identical, the primitive succeeds.

The 4 x 4 submatrices shown in Figure 7-2(a) were used to

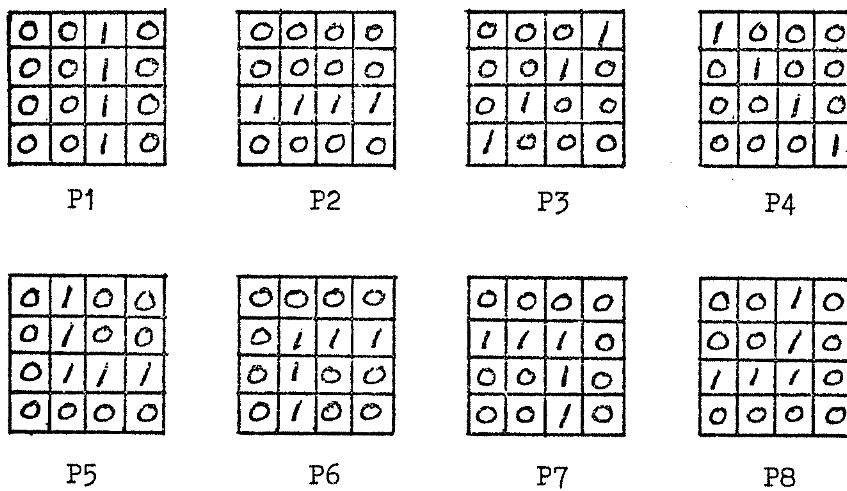


(a) A typical memory structure built up from letter primitives

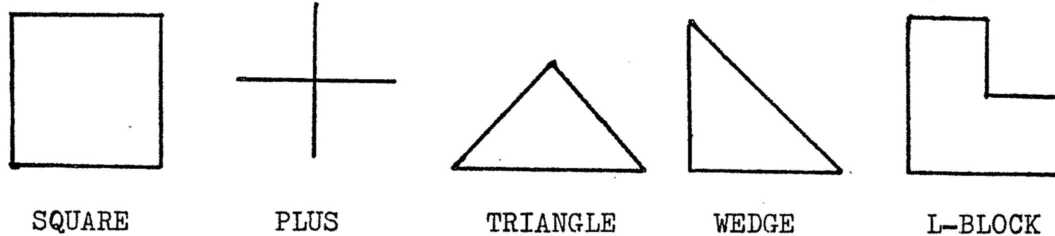


(b) A letter described in terms of lower level primitives

Figure 7-1. Letters used as primitives.



(a) 4 x 4 submatrices used by HPL



(b) Patterns recognized with primitives P1 through P8

Figure 7-2. Pattern recognition using submatrix primitives.

recognize the patterns in Figure 7-2(b), which were input as 10 x 10 matrices. The patterns were recognized correctly after one learning trial.

Submatrices are used as follows: Suppose [P1 (2,3) 1.] is to be computed. At location (2,3) in the input, an exact template match between P1 and the input is sought. If such a match is not found, a similar comparison between P1 and the input will be made for each location within a neighborhood (±2) of (2,3), because of the TOLERANCE parameter. If a match succeeds at any location in the neighborhood, P1 is FOUND; otherwise, it is NOT-FOUND.

Although not illustrated in Figure 7-2, submatrices of various dimensions may be used at the same time. For example, P1 could be 6 x 3 and P2 4 x 5. However, the SIZE parameter currently serves no useful purpose with submatrices, for any particular submatrix has only one fixed size, determined by its dimensions. (In Chapter 9 we discuss the use of a coarse and fine grid representation of the input, giving meaning to SIZE with submatrices.)

There are many ways in which submatrices can be made more powerful, for example, permitting a 90% match rather than a perfect match, or allowing a -1 submatrix cell to indicate that the cell should be ignored in the matching process. However, the well-formed patterns of Figure 7-2 did not require more sophisticated techniques. Because the LISP system we used was somewhat

inefficient in performing matrix operations*, most patterns were recognized using the less expensive LINE primitives, rather than submatrices. Our main intent in using submatrices was to demonstrate that it could actually be done, showing that HPL is indeed independent of the primitives used.

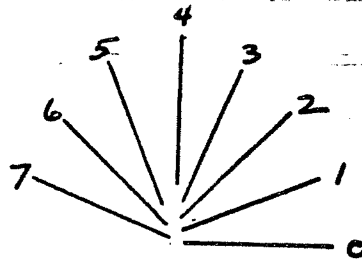
1.3 LINE Primitives

LINE primitives have characterizer names of the form LINE-i, where i is an integer between 0 and 7 designating the discrete slope of the line. Figure 7-3(a) shows the slope corresponding to each integer. An integer represents any slope within a fixed range; for example, LINE-0 is a line which meets the X axis at an angle between $\pm 11\frac{1}{4}^{\circ}$. A line is designated [LINE-i LOC SIZE], where LOC and SIZE determine the position of the associated rectangle surrounding the line, as previously explained.

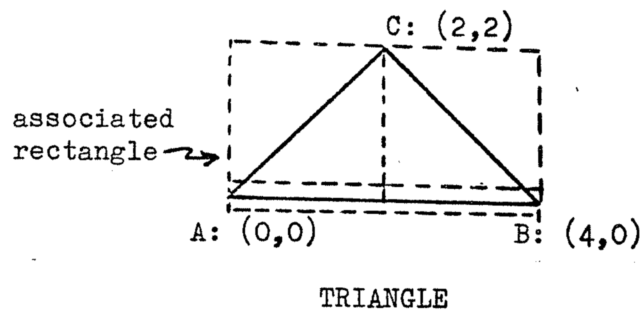
The input scene was presented as a list of vertices, each named by a letter of the alphabet for convenient inputting, the X,Y coordinates of the vertex, and a list of those vertices connected to it by a straight line. Figure 7-3(b) shows the representation of a triangle in this format. We assume that a preprocessor, for example, a line-following routine, could convert a simple line drawing into that format (an assumption also made by Guzman [1968]). This input format is converted into a set of

---

*Our LISP system lacks an ARRAY feature. See Chapter 9 for a discussion of computer language facilities we would like to have available.



(a) Slope, represented by an integer between 0 and 7



TRIANGLE

[[A 0. 0. (B,C)], [B 4. 0. (A,C)], [C 2. 2. (A,B)]]

INITIAL INPUT FORMAT

[[LINE-0 (0.,0.) 4.0], [LINE-5 (2.,0.) 2.0], [LINE-3 (0.,0.) 2.0]]

PREPROCESSED PATTERN: a list of instances

(b) A triangle, its representation in the initial input format, and its preprocessed representation.

Figure 7-3. An example using LINE primitives.

instances of the form [LINE-i LOC SIZE], which are placed directly on the COMPLETED list. HPL never accesses the initial input format directly. Figure 7-3(b) also shows the preprocessed LINE-i primitives corresponding to the initial input representation. LINE primitives are preprocessed because a hypothetical line-following routine would normally output all lines it found in a certain region, rather than determining if there were a specific line of a particular slope and size at a certain location.

1.4 The Relationship between LINE and Submatrix Primitives

Most computer runs of HPL have used LINE primitives rather than submatrices for efficiency, as already stated. We have also indicated that HPL is independent of the primitives used. Certainly, recognition may be more accurate or more efficient with some types of primitives than with others, and this may be a function of the patterns being recognized, but this does not affect HPL's structure. Similarly, the recognition examples presented below are to some degree independent of the primitives used, despite the fact that LINE primitives were used with all of them. Some primitives might be more appropriate with certain pattern types than others, but the general part/whole pattern relationships reflected in the memory structure is not dependent on the type of primitive. In this section, we discuss the relation between LINE primitives and submatrices.

Submatrices are more general than LINE primitives, for a submatrix of any configuration of 0's and 1's may be used, each a template for a different shape. Submatrices are amenable to learning, as well; for example, a novel submatrix that appears in an input may be added to the repertoire of primitives (as in Uhr and Vossler[1961]). The advantages of submatrices are counterbalanced by the large number of possible submatrices, many of which are of little value. A LINE primitive is a more specific, "higher-level" primitive than a submatrix. (In Chapter 8, we discuss other higher-level primitives that might be used.) A higher-level primitive is a particular function the program designer considers important in recognition. This eliminates the potentially costly search for good lower-level primitives, but requires reliable intuitions in choosing good primitives.

Some higher-level primitives may be built up or at least approximated by lower-level primitives. LINE primitives might be approximated by submatrices representing straight lines of each discrete slope, as in Figure 7-2(a), where submatrices P1 to P4 represent straight line segments of 0°, 45°, 90°, and 135° angles with respect to the X-axis. Such primitives do not duplicate LINE primitives, which indicate where lines begin and end, and whose length may exceed the size of a submatrix. Submatrices representing various vertices, such as P5 to P8 in Figure 7-2(a), indicate line endpoints, as well as give information about the type of endpoint. In other words, to exactly represent a LINE instance

may require many submatrix instances. (Representing a LINE in terms of submatrices is analogous to representing letters in terms of LINEs, rather than as primitives themselves, as in Section 1.1.)

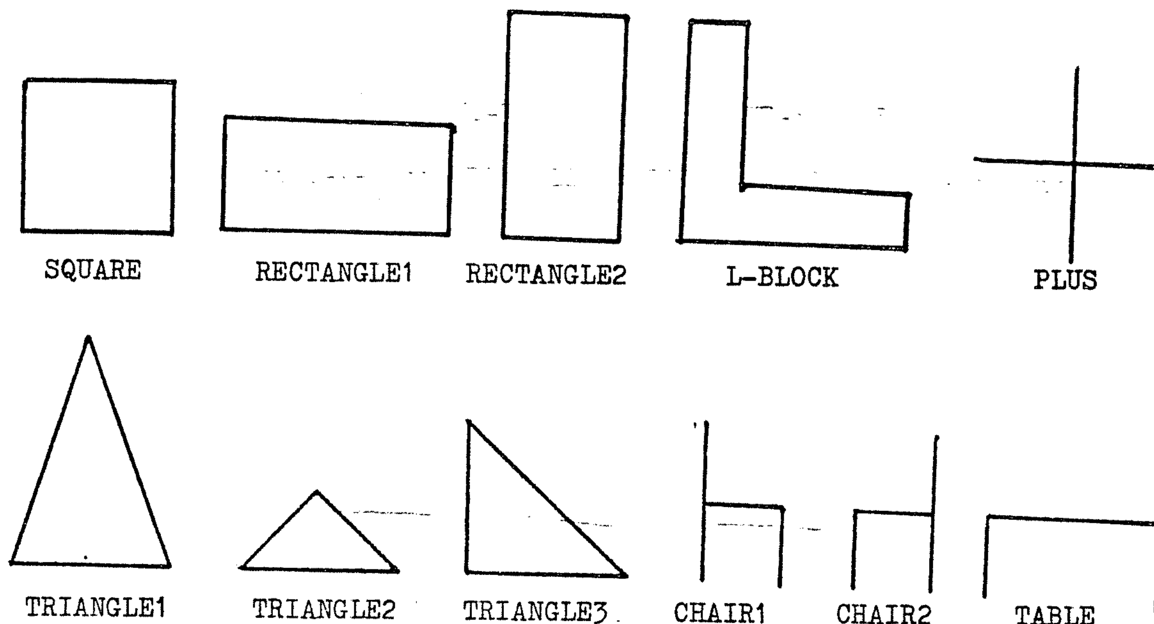
It is easy to think of higher-level primitives that could not easily be approximated by submatrices, such as a primitive that computes the number of straight lines in a pattern.

In order to recognize more realistic patterns than is currently possible, the generality of lower-level primitives such as submatrices may be essential. Higher-level primitives can probably be written which work effectively on idealized patterns, yielding more immediate results than long computer runs attempting to discover good submatrices. As to the value of using idealized patterns at all, we believe that important issues, such as the recognition of subpattern/pattern relationships and the question of context, may be investigated with highly simplified patterns of the kind HPL recognizes.

2. Simple Pattern Recognition

Figure 7-4(a) shows a set of simple patterns recognized correctly after one learning trial with LINE primitives. These patterns are "simple" because they do not contain any patterns as subparts. When a pattern is recognized, its NAME, LOCation, and SIZE are output.

Figure 7-4(b) shows the memory representation of the goal



(a) Simple patterns recognized by HPL

```
Description-set of SQUARE:                         THRESHOLD = 30
 [ [(LINE-4 (1. 0.) 0.1) 5], [(LINE-0 (0. 1.) 1.0) 5],
   [(LINE-4 (0. 0.) 0.1) 5], [(LINE-0 (0. 0.) 1.0) 5],
   [(C1 (0. 0.) 0.1) 5], [(C2 (0. 1.) 1.0) 5] ]

Description-set of C1:                             THRESHOLD = 15
 [ [(LINE-4 (0. 0.) 1.0) 10], [(LINE-0 (0. 0.) 10.) 10] ]

Implication-set of C1:              USE = 12, VALUE = 0.60
 [ [(PLUS (-1.0 -1.0) 20.0) 2], [(L-BLOCK (-1.0 -1.0) 13.3) 2],
   [(L-BLOCK (0. 0.) 10.0) 2], [(CHAIR1 (0. -1.0) 10.0) 2],
   [(TRIANGLE3 (0. 0.) 10.0) 2], [(SQUARE (0. 0.) 10.0) 2]

Description-set of C2:                             THRESHOLD = 15
 [ [(LINE-0 (0. 0.) 1.0) 10], [(LINE-4 (1.0 -1.0) 0.1) 10]

Implication-set of C2:              USE = 8, VALUE = 0.57
 [ [(PLUS (0. -1.0) 2.0) 2], [(CHAIR1 (0. -1.0) 1.0) 2],
   [(CHAIR2 (0. -1.0) 1.0) 2], [(SQUARE (0. -1.0) 1.0) 2]
```

(b) Memory representation of SQUARE and two compounds

Figure 7-4. Simple patterns recognized with LINE primitives.

characterizer SQUARE after HPL had been presented each pattern twice. Also shown are the characterizers representing compounds C1 and C2, descriptors of SQUARE. Each compound, in turn, has two LINE primitive descriptors.

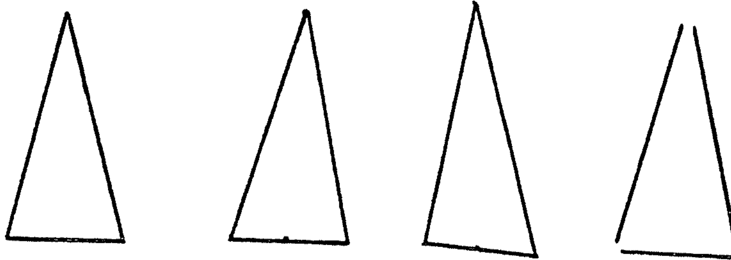
3. The Representation of a Pattern

Three patterns in Figure 7-4(a) were given pattern names TRIANGLE1, TRIANGLE2, and TRIANGLE3. Why were they not all given the same name, TRIANGLE? They can be, but only indirectly through explicit training.

First we must describe what HPL will recognize as an example of TRIANGLE1. Ignoring compounds (which would complicate but not substantially change the situation), the description-set of TRIANGLE1 would be represented as shown in Figure 7-5(a). Because of the TOLERANCE factor, any of the triangles in Figure 7-5(b) will be recognized as TRIANGLE1. (Note that one pattern is not even connected, since LINE primitives do not test for connectedness.) Each LINE instance individually matches the appropriate descriptor in the description-set of TRIANGLE1 within the TOLERANCE. Suppose now that the three triangles which we have called TRIANGLE1, etc., were each presented to HPL as an example of TRIANGLE but were not named individually. The description-set of TRIANGLE shown in Figure 7-5(c) would result. The characterizers that would have been separate descriptors of TRIANGLE1, etc., are now all descriptors of TRIANGLE. After sufficient weight adjustment, an

[[LINE-0 (0. 0.)], [LINE-3 (0. 0.)], [LINE-5 (0.5 0.)]]

(a) Description-set of TRIANGLE1 (SIZE and DWT have been ignored for clarity)



(b) Several patterns which will all be recognized as examples of TRIANGLE1

Possible description-set of TRIANGLE, assuming each DWT = 5, and THRESHOLD = 15:
[[LINE-0 (0. 0.)], [LINE-3 (0. 0.)] [LINE-5 (0.5 0.)],
[LINE-2 (0. 0.)], [LINE-6 (0.5 0.)], [LINE-4 (0. 0.)],
[LINE-6 (0. 0.)]]

(c) A possible description-set for TRIANGLE when TRIANGLE1, etc., have not been learned separately

(d) A composite nonsense pattern which will satisy the above description-set

[[TRIANGLE1 (0. 0.)], [TRIANGLE2 (0. 0.)] [TRIANGLE3 (0. 0.)]]

(e) Description-set of TRIANGLE when TRIANGLE1, etc. have been previously learned

Figure 7-5. Various ways of describing a triangle.

occurrence of any of the triangles would cause the THRESHOLD to be exceeded and TRIANGLE recognized, as desired. Unfortunately, a nonsense shape, such as in Figure 7-5(d), containing enough descriptors to exceed THRESHOLD, would also be recognized as TRIANGLE.

We have approached the problem by first teaching HPL as separate patterns TRIANGLE1, TRIANGLE2, and TRIANGLE3. Then when one of these patterns is recognized, HPL is given feedback that TRIANGLE is present. This produces the goal TRIANGLE shown in Figure 7-5(e), which contains goals TRIANGLE1, etc., as descriptors. In effect, TRIANGLE is described as a disjunctive concept whose instances are TRIANGLE1, etc.

It would be desirable if HPL could decide, without feedback, to form compounds of dissimilar instances of a pattern concept. Then, TRIANGLE1, TRIANGLE2, etc., would be represented as compounds describing the goal TRIANGLE, rather than as goals themselves. HPL's compound formation mechanism is a step in this direction, but a more powerful algorithm is probably needed.

Teaching TRIANGLE1 as an instance of TRIANGLE illustrates a simple class-forming ability inherent in HPL's structure. Suppose HPL has learned HOUSE, BARN, STORE, etc. HPL can then be taught that each such pattern is a BUILDING. If HPL has learned individual letters of the alphabet, each letter can be taught as an example of the pattern class ALPHABET-LETTER. Once a class

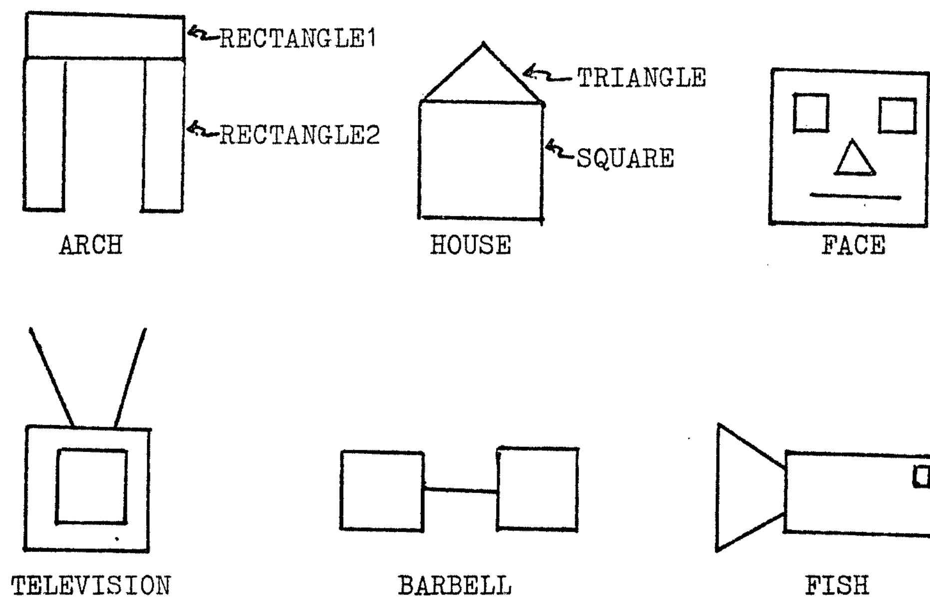
name has been learned, it may be used as a descriptor of other patterns. For example, HOUSE can be defined as TRIANGLE and SQUARE, rather than the more specific TRIANGLE1 and SQUARE.

4. Recognition of Complex Patterns

Figure 7-6(a) shows a set of "complex" patterns successfully recognized by HPL, which contain other patterns as subparts. The subparts are also recognized. HOUSE, for example, contains sub-patterns SQUARE and TRIANGLE. In attempting to recognize HOUSE, HPL will first output the names SQUARE and TRIANGLE, their locations and sizes, and then HOUSE, its location and size.
A subpattern/pattern relationship between goals G1 and G is represented in memory by a downward path from G to G1. Usually, G1 is a descriptor of G, as in the description-sets of HOUSE and ARCH shown in Figure 7-6(b).

There is no theoretical limit in HPL on the complexity of patterns that can be recognized, for example, subpatterns of sub-patterns of patterns. Practically, of course, the more complex the pattern, the greater the time required for recognition. For this reason, we have only used patterns of sufficient complexity to demonstrate HPL's ability. It is important to realize, however, that there is no fundamental difference between recognizing a TOWER composed of three ARCHes, and recognizing a single ARCH composed of three RECTANGLEs (as in Figure 7-6(a)).

In teaching HPL complex patterns, subpatterns need not be



(a) Complex patterns recognized by HPL

Description-set of HOUSE:
  [[SQUARE (0. 0.) 1.0], [TRIANGLE (0. 1.) 1.]]

Description-set of ARCH:
  [[RECTANGLE2 (0. 0.) 1.0], [RECTANGLE2 (3. 0.) 1.0],
    [RECTANGLE1 (0. 4.) 1.0]]

(b) Description-sets of HOUSE and ARCH (ignoring DWTs)

Figure 7-6. "Complex" patterns recognized by HPL with LINE primitives.

learned before the whole pattern is learned. For example, HOUSE in Figure 7-6(a) can be learned before TRIANGLE and SQUARE have been learned. The description-set of HOUSE will then contain only LINE primitives and compounds. Suppose HOUSE is learned, and then TRIANGLE and SQUARE are taught. Then the next time HOUSE is presented as a pattern, SQUARE and TRIANGLE will also be recognized, but not as subpatterns of HOUSE. The link-forming algorithm will then add the goals representing SQUARE and TRIANGLE to the description-set of HOUSE. The new information that SQUARE and TRIANGLE are subpatterns of HOUSE is integrated into the existing knowledge about HOUSE.

In recognizing FACE, shown in Figure 7-6(a), HPL also recognizes subpatterns SQUARE and TRIANGLE. SQUARE represents EYE, TRIANGLE is NOSE, and the straight line is MOUTH. However, we do not want HPL to output EYE when _any_ SQUARE is recognized, but only when the particular SQUAREs in FACE are recognized. This is the context problem. Note that FACE will be recognized successfully with SQUARE aiding that recognition. The context problem is that SQUARE should _therefore_ be recognized now as EYE. A modification of HPL to handle this problem is discussed in Chapter 8. An unconvincing temporary solution would require that EYE contain details allowing it to be differentiated from SQUARE regardless of whether FACE was also recognized.

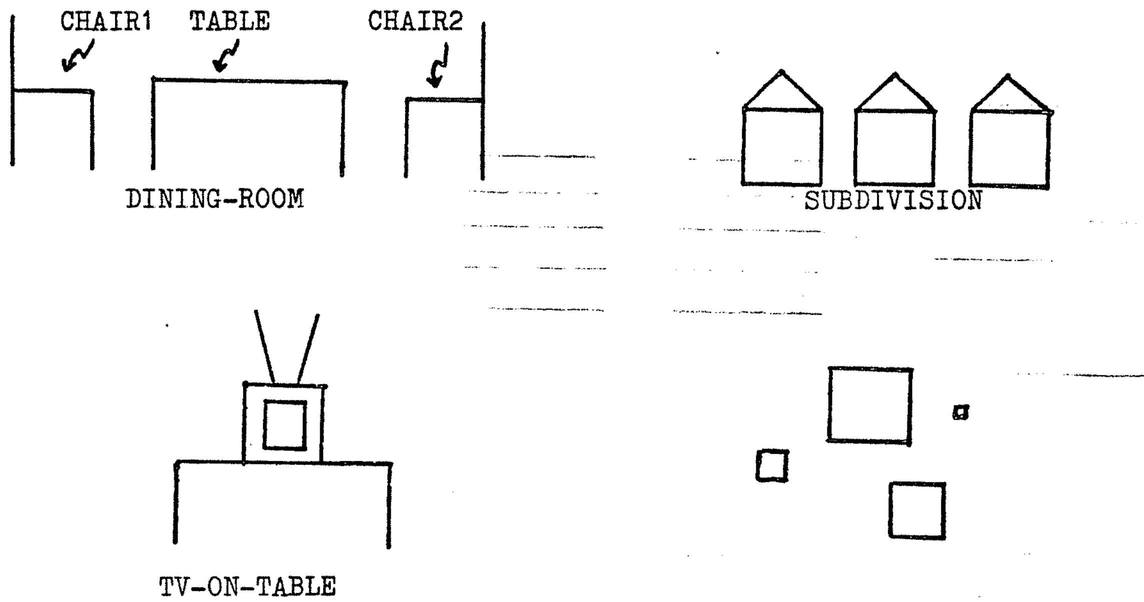
5. Scene Analysis

Figure 7-7(a) shows some simple "scenes" recognized by HPL. Recognition of scenes is identical to recognition of complex patterns; memory structures are identical. Each pattern and sub-pattern name is output, along with its location and size. For example, in the scene containing several SQUAREs of different sizes, HPL outputs the name of each SQUARE, its location, and size. A scene may be given an overall name, such as DINING-ROOM. Figure 7-7(b) shows the memory representation of DINING-ROOM.

We do not believe there should be a distinction between scene analysis and recognition of complex patterns. It is difficult, in fact, to define what differentiates scene analysis from complex pattern recognition or what processes might be more applicable to one than to the other. They both contain the part/whole relationships that HPL deals with. For example, recognition of a nose-like object may suggest a face, which in turn suggests eyes, mouth, etc., in the same way that recognition of a couch may suggest a living room, which in turn suggests a lamp, a chair, and so forth. One might argue that scene analysis should involve recognition of global features; this is probably true, but would seem to apply as much to object recognition. (We discuss "general to specific" recognition in Chapter 9.)

6. Overlapping Patterns

Figure 7-8(a) shows a simple example of overlapping patterns.



(a) Scenes recognized by HPL

[[CHAIR1 (1. 0.) 0.8], [TABLE (1.5 0.) 1.0], [CHAIR2 (5.0 0.) 0.8]]

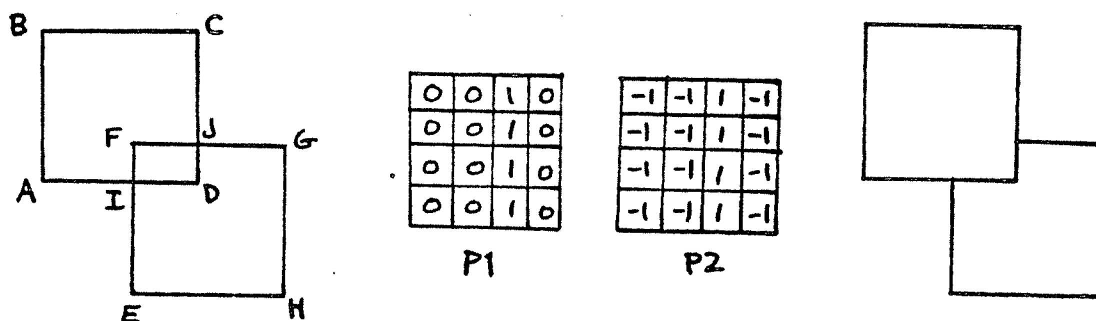
(b) Description-set of DINING-ROOM scene

Figure 7-7. Scene recognition.

Whether the scene will be recognized as two SQUAREs depends on how primitives are computed. LINE primitives assume the input has been preprocessed by a line-following mechanism. Depending on its design, that mechanism might find one line between A and D, or two lines, one between A and I and one between I and D. In terms of LINE primitives, this is a question of how we input the pattern, including or ignoring vertices I and J. In terms of submatrices, it is a question of whether a primitive such as P1 in Figure 7-8(b) superimposed on line EF with I in between will be FOUND or not. As implemented, it would not be FOUND, but one could easily define a submatrix such as P2, in which -1 cells were ignored, that would be FOUND.

Should line AD be considered one line or two? Nothing in HPL requires choosing between the two alternatives. For example, the pattern can be presented in the input format so that lines AD, AI, and ID will all be FOUND. Then both SQUAREs will be recognized, as well as rectangle IDJF. In other words, overlapping patterns pose no problem to HPL, as long as all necessary primitives are FOUND, and the simplest way to accomplish that is to allow all possible interpretations of the input.

It is much easier to recognize overlapping patterns, however, than it is to recognize occluded patterns, as in Figure 7-8(c), discussed in Chapter 8.



(a) Overlapping squares (Letters identifying intersections are not part of the scene.)

(b) Two variants of a submatrix primitive

(c) Occluded squares

Figure 7-8. Overlapping patterns.

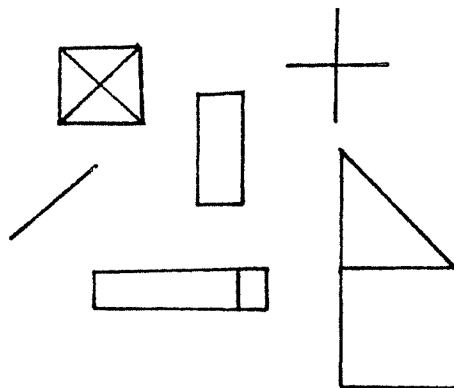


Figure 7-9. The needle in the haystack: find all occurrences of TRIANGLE in the scene.

7. Attending to a Pattern: The Needle in the Haystack

In the "needle in the haystack" problem, HPL is instructed to find all occurrences of a specified pattern within the input scene. For example, HPL successfully found the TRIANGLE in Figure 7-9. HPL easily handles this problem by use of the VALUE parameter of characterizers. Normally, all goals are given the same VALUE, 1.0. Here, the desired goal is given a higher VALUE, such as 10. The VALUEs of all descriptors of the goal, and their descriptors, etc., are then recomputed, since they depend on the VALUE of the goal. Since the active weight depends on VALUE, the recognition algorithm is biased in the desired direction; the greater the new VALUE, the greater the bias. In other words, HPL will give the desired pattern the most attention. The VALUE of more than one goal can of course be changed as well. Hence, HPL can be asked to find some set of patterns.

In the same manner, HPL can look for a desired pattern class. If letters of the alphabet were also taught as examples of pattern class LETTER (as discussed in Section 3), and numerals taught as examples of pattern class NUMERAL, then recognition could be biased toward recognizing LETTERs or NUMERALs. Similarly, in a "robot" or "simulated organism" framework, recognition could be biased to attend to patterns of class GEOMETRIC-SOLID or ANIMAL.

A frequently cited example of human pattern recognition

beyond the ability of mechanical pattern recognition is the "face in the crowd". Someone looking at a large group of faces may suddenly recognize one person in the crowd. Note that the needle in the haystack problem is not the same as the face in the crowd problem. The former directs recognition toward a particular pattern; the latter involves recognition of an unexpected pattern. In fact, it is the unexpectedness of any particular face that makes the problem interesting. We discuss this important problem in Chapter 8.

8. Evaluation of Results

HPL successfully learned and recognized the types of pattern shown in this chapter, usually after only one presentation of each pattern. This validates the general structure of the memory and learning mechanism. The compound mechanism, however, is largely unvalidated, since by its nature it requires extended learning, which we have avoided because of computing costs.

Since LINE primitives are assumed to be "preprocessed", it is meaningless to ask how efficiently the recognition algorithm would compute them. Submatrices, however, were computed serially. After first learning the patterns in Figure 7-2, HPL recognized them using approximately one third of the total possible number of primitive and location combination; all possible primitive and location combinations would have been computed in a simulated parallel program. This suggests that the recognition algorithm

is able to compute primitives efficiently.

The patterns shown in Figure 7-4 were recognized after an average of 3.9 seconds of processing time (.8 seconds was the shortest time, 5.8 the longest). Although processing times are not especially meaningful because of the many factors involved (the computer used, the language and its implementation, etc.), we felt they were encouraging but larger than expected. It appears that a large part of the processing time is spent in creating instances, which we had not foreseen. There is a simple mechanism that might significantly reduce this time. Rather than creating a complete instance consisting of description-set, implication-set, etc., as defined in Chapter 4, a much simpler instance could be created, consisting only of the characterizer defining the instance, [CHAR LOC SIZE], and instance weights (AWT, FOUNDWT, etc.). A complete instance would only be created in the comparatively infrequent case that the AWT was sufficiently large.

One might assume that the recognition time for a complex pattern such as HOUSE would be greater than the sum of the recognition times for each of its subpatterns TRIANGLE and SQUARE, when presented alone. This was not the case. HOUSE, for example, was recognized in approximately the same time as required to recognize SQUARE alone.

The reason is not hard to see. Since LINE primitives are presented in a coded format requiring little primitive processing, recognition times primarily reflect costs involved in memory search,

instance formation, etc. When HOUSE was presented, TRIANGLE was recognized first; TRIANGLE is one of the easier patterns in HPL's repertoire, having few descriptors in common with other patterns. Since few higher-level patterns containing TRIANGLE as a subpattern had been learned, HOUSE was given a high active weight. HOUSE was soon investigated and an instance of SQUARE was formed with a high active weight. HPL then directly found SQUARE at the appropriate place, and hence HOUSE.

SQUARE, when presented alone, is one of the more difficult patterns for HPL, since its descriptors are common to several patterns. However, as part of HOUSE, it is more easily recognized, for recognition proceeds from TRIANGLE to HOUSE to SQUARE, and back to HOUSE. HPL's recognition algorithm encourages such bottom-up/top-down behavior because of its efficiency.

We tested the needle in the haystack ability as follows: HPL was first presented the scene in Figure 7-9 without being requested to look for a specific pattern. After 30 seconds, recognition was terminated. HPL had recognized some of the patterns in the scene, but not TRIANGLE. Then HPL was given the same scene but was requested to look for TRIANGLE. HPL successfully recognized TRIANGLE after approximately 4 seconds. By giving TRIANGLE a high VALUE, its descriptors also had high VALUEs and were given high priority in recognition. We find these results very encouraging.

CHAPTER 8

EXTENSIONS OF HPL

1. Introduction

In this chapter, we discuss extensions to HPL which improve its recognition ability. For the most part, these extensions are not vague conjectures, but instead concrete proposals that could be embodied in computer code. In the following chapter, we discuss less concrete extensions.

Many of the suggestions may have a quantitative rather than qualitative effect on the type of pattern HPL recognizes, increasing the variability allowed in a pattern and the tolerance for error. The suggestions concerning relations (Section 4) and context (Section 5) may have more far-reaching effects, increasing the descriptive power of memory.

2. Expanding the Structure of the Characterizer

In Chapter 4, we explained that the description-set, implication-set and miscellaneous parameters which constitute a characterizer are stored in memory associated with the characterizer name. We also explained that a descriptor and implicand each contain a characterizer reference of the form [NAME LOC SIZE]. In addition, an instance is identified by a characterizer reference. In this section we shall discuss ways to expand this structure. A characterizer reference will be considered in a more general form:

[NAME ARG1 ARG2...], where ARGi is an argument or attribute of the characterizer reference.

2.1 Absolute and Relative Attributes

Before discussing extension of the characterizer, we should briefly discuss the storage of relative and absolute information in HPL's memory. Location and size are stored in permanent memory _only_ as relative quantities, for the characterizer references in descriptors and implicands have only relative arguments. These relative arguments are converted to absolute arguments stored within instances during recognition, as described in Chapter 5.

When might absolute location have meaning in the input? HPL's input is supplied by a trainer, who decides what aspects of a pattern are meaningful. Suppose the trainer always presented patterns that filled a 10 x 10 region, or that a preprocessor expanded inputs to fill a 10 x 10 region. The absolute location of a characterizer would then be meaningful. For example, if the LETTER-A always filled a 10 x 10 region, a submatrix P1 representing the vertex at the top of the A would succeed near the upper middle of the input area, near (5,10). Thus, if P1 were found at or near absolute location (5,10), LETTER-A would be a possible input. Moreover, if P1 were found at (2,4), LETTER-A would not be a possibility, since the top part of A would not occur at such a location.

In contrast, suppose HPL finds P1 at (2,4). Since HPL

assumes location is relative, it cannot rule out the possibility that LETTER-A is present in the input with its top vertex at (2,4). In not assuming absolute location, HPL must consider more possibilities in the input. Conversely, preprocessing so that absolute location is meaningful reduces the number of possibilities. However, such preprocessing eliminates the ability to determine the location of a pattern (since it always occurs in the standard position). Similarly, it eliminates the ability to determine the size of an input (since the size is also always the same.)

HPL must be able to determine the locations and sizes of subpatterns in order to recognize complex patterns containing subpattern/pattern relationships; a simple example is a pattern containing several rectangles of varying sizes and locations as subpatterns. Furthermore, a preprocessor cannot expand each subpattern to fill the input array. Suppose a HOUSE consisting of a TRIANGLE and SQUARE is input. The program cannot assume that TRIANGLE has been expanded to fill the input, for that is impossible if HOUSE has been expanded.

Although HPL does not store absolute location and size, it does, when using LINE primitives, store absolute slope. Arguments in a characterizer reference must be relative, and so absolute slope is incorporated into memory by including it as part of the characterizer name: LINE-i. If a LINE of slope i is FOUND, the primitive LINE-i in memory will be accessed. Hence, HPL assumes that absolute slope is meaningful. For example,

suppose HPL finds a vertical line and expects a HOUSE composed of TRIANGLE and SQUARE. HPL will consider two possibilities: the vertical line may be the left hand "wall" or the right hand wall of HOUSE (i.e., one of the two vertical sides of SQUARE). It will not consider the possibility that the vertical line is the floor of HOUSE lying on its side. In effect, HPL assumes the trainer or preprocessor has oriented patterns so that absolute slope is meaningful. As shown above, this reduces the number of possibilities that must be considered. We felt that adding relative slope in addition to relative location and size would increase the number of possibilities to the point of making HPL too expensive to test.

To summarize, absolute information <u>may</u> be meaningful, depending on what assumptions are made about the input. If it is meaningful, it is valuable to allow it to be stored within memory.

HPL's method of incorporating absolute slope within the NAME of a characterizer is <u>ad hoc</u> and makes it awkward to add additional absolute attributes. A more unified approach should allow each argument ARGi in the characterizer reference [NAME ARG1...] to be either relative or absolute. Absolute slope would thus be represented as an argument. In fact, absolute and relative slope could both be represented as separate arguments. In Section 2.4, we discuss associating weights with individual arguments. With that mechanism, a characterizer reference could indicate that absolute slope was <u>usually</u> important in a certain pattern. For

example, HOUSE might usually be expected in an upright position. However, the much less likely possibility of HOUSE lying on its side could also be entertained, with a correspondingly low active weight.

2.2 Multi-Valued Functions

In HPL primitives are either FOUND or NOT-FOUND in the input. With the LINE type of primitive in mind, let us examine a new primitive called VERTEX which computes the type of vertex or intersection of two or more lines. Vertices are places of high information content in a pattern and should therefore make good primitives. Figure 8-1 shows several types of vertices, called TYPE-A, TYPE-B, etc. Suppose HPL were to look for a TYPE-A vertex at a certain location, expressed as [VERTEX TYPE-A LOC], where TYPE-A is an absolute argument as in the preceding section. If there were a specific function to compute each particular vertex type, i.e., a TYPE-A function, a TYPE-B function, and so forth, then HPL could handle the situation without modification. However, it is more reasonable to assume that a general VERTEX function would, given a location, return as a value the type of vertex found. If HPL issued the primitive function request [VERTEX TYPE-A LOC], and a vertex of TYPE-B was FOUND, it would be wasteful to merely return a NOT-FOUND value to the function request, ignoring the information that a TYPE-B vertex had been FOUND. Within HPL's existing structure, it would be

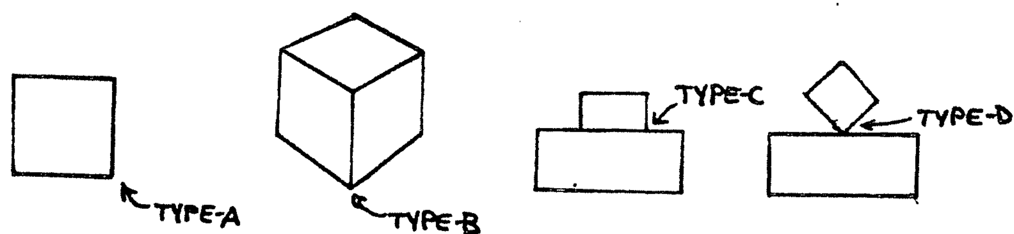
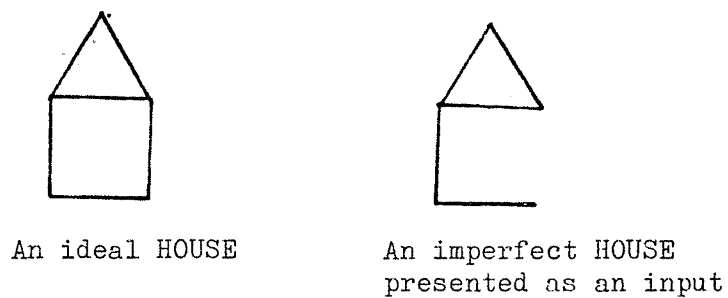


Figure 8-1. Several types of vertex that might be recognized by a vertex classifier.



An ideal HOUSE

An imperfect HOUSE presented as an input

Figure 8-2. An imperfect HOUSE and an ideal HOUSE

possible both to return a NOT-FOUND value to the instance [VERTEX TYPE-A LOC] and also to create a new instance [VERTEX TYPE-B LOC] marked as FOUND and added to the COMPLETED list. In short, with little change to HPL's structure, primitives can return values other than FOUND/NOT-FOUND by expressing them as characterizer arguments.

2.3 Comparison of Characterizers

The previous section raises the issue of comparing characterizer arguments. A vertex instance with argument TYPE-A was sought but an instance with argument TYPE-B was FOUND. A TYPE-B argument does not satisfy the request for a TYPE-A argument, that is, TYPE-A does not "match" TYPE-B. For greater generality, the _partial_ matching of two arguments should be considered.

The LOC argument is an example. Suppose [NAME LOC1...] is sought and [NAME LOC2...] is FOUND; that is, some feature sought at LOC1 is found instead at LOC2. In HPL, LOC1 and LOC2 only match successfully if they are closer together than the TOLERANCE parameter. An alternative is to allow partial matching of LOC1 and LOC2, computing a match weight between 0. and 1. which depends on the degree of closeness.

Before a new instance [NAME ARG1...] is created, a comparison is made to determine whether a similar instance already exists (e.g., in determining the MATCH fields in the description-set of an

instance). In this comparison, an exact match within TOLERANCE is required of each argument. For example, if some feature is sought but has already been FOUND at a similar location with a similar size, it will not be sought again. We can extend the match weight (MATCHWT) concept to the entire characterizer reference, computing a total MATCHWT between 0. and 1. as a function of MATCHWTs between arguments. If this total MATCHWT is above some fixed criterion, e.g., .8, the two instances match. This slightly increases the flexibility of the matching algorithm: two instances might match whose locations are slightly beyond the previous TOLERANCE factor, but whose sizes match closely, or vice versa.

A more powerful use of MATCHWT, however, is in the computation of the cumulative description weight of an instance. Rather than summing the DWT of each descriptor, DWT * MATCHWT would be summed. In other words, those descriptors which did not as closely match what was actually FOUND would contribute less to CUM-DWT. This allows greater variability in an input, and at the same time, takes into account the amount of variability in deciding what is present. Within the learning process, the amount of DWT adjustment could also depend on MATCHWT.

Essentially, we have replaced a discrete mechanism ("match" or "no match" for arguments and characterizers) with a continuous one. This can be carried one step further, replacing the FOUND/

NOT-FOUND concept with a continuous mechanism. The implications of this final step are discussed in Section 3. Before doing so, however, we shall discuss the issue of weights associated with characterizer arguments.

2.4 Weighted Characterizer Arguments

We can further increase the power of the memory by associating weights with characterizer arguments to indicate their relative importance. The weights would function much like DWTs. The structure of a characterizer reference would then be: [NAME (ARG1 WT1) (ARG2 WT2)...]. The argument weights would enter into the function that computes the MATCHWT between characterizers suggested in the previous section. (Such weights are discussed in Becker[1970].) For example, a descriptor of some pattern might be [LINE (SLOPE1 5) (LOC1 2)], indicating that the slope of the LINE is more important than its location.

Argument weights are only useful if a learning algorithm can successfully adjust them to meaningful values. The same type of approach used for adjusting DWTs probably can be used, downweighting closely matching arguments of a descriptor in an instance incorrectly FOUND, upweighting other arguments, and so on. Further experimentation is necessary, however, to determine whether the weights will converge to meaningful values, given the added complexity of the memory.

3. Partially Found Characterizers

In Chapter 5, we explained that FOUNDWT (defined as $(\text{CUM-DWT} / \text{THRESHOLD})^2$), which is used in computing the active weight, is a measure of how close the CUM-DWT of an instance is to THRESHOLD. An instance is only considered FOUND, however, when CUM-DWT reaches THRESHOLD, which implies that FOUNDWT is 1.0; this is an important simplification. The alternative we shall discuss here is to eliminate the idea of FOUND instances but retain FOUNDWT as a measure of how confident HPL is that the instance is present. This would allow HPL to make judgements on the basis of less perfect information than now required.

Suppose the imperfect pattern HOUSE of Figure 8-2 were presented to the current version of HPL, which had been trained to expect HOUSE as a SQUARE and TRIANGLE with the proper relative locations and sizes. If the input is to be recognized as HOUSE, both TRIANGLE and SQUARE must be FOUND. TRIANGLE will be FOUND, but since SQUARE has a side missing, it may not be FOUND. Whether it is FOUND depends on the relation between the DWTs of the descriptors of SQUARE and its THRESHOLD. The lower the THRESHOLD, the fewer descriptors are needed to "fire" SQUARE; but this depends on HPL's training. If perfect SQUAREs are always presented, HPL will have a low tolerance toward deviation from the ideal. If HPL is trained with less than perfect SQUAREs, the three-sided SQUARE will have a greater chance of being recognized. If SQUARE is FOUND, then HOUSE will be also, but if it is not, then neither will HOUSE.

It would be preferable if HPL's recognition of imperfect patterns were less dependent on the training sequence.

Allowing partially found instances lessens the dependence of HPL on the training sequence and increases the flexibility of HPL. If SQUARE is always presented as a perfect pattern, then a perfect SQUARE will have a FOUNDWT near 1. But if only 3 sides are present, SQUARE will be partially found; that is, it will have a FOUNDWT less than 1., such as, .75. With the mechanism to be described, this FOUNDWT will enter into the CUM-DWT of HOUSE, which will also have FOUNDWT less than 1., such as .8. Without this mechanism, SQUARE would be NOT-FOUND and would <u>not</u> contribute to the CUM-DWT of HOUSE. If HPL's response criterion is to output goals having the highest FOUNDWTs, HPL might output HOUSE, .8, TRIANGLE, 1.0, and SQUARE, .75. In the FOUND/NOT-FOUND approach, HPL would only output TRIANGLE. If HPL were trained with less than perfect examples of SQUARE, the three-sided SQUARE would have a higher FOUNDWT, and so would HOUSE, but the pattern names output by HPL would not change. The FOUND/NOT-FOUND approach is discontinuous, for when FOUNDWT reaches 1.0, behavior changes greatly. With the proposed approach, behavior is continuous with respect to FOUNDWT.

Observe that with this proposed mechanism, a pattern may be output before its subpatterns are. In recognizing FACE, each subpattern, EYE, NOSE, etc., may be partially recognized, e.g., with

a low FOUNDWT. The cumulative effect, however, may give FACE a high FOUNDWT, causing it to be output. At this point, recognition could terminate, or it could continue to more completely recognize the subpatterns.

The most important aspect of the mechanism for handling partially found characterizers is as follows. Suppose descriptor D of instance X is matched to an instance D'. In the current version of HPL, DWT(D,X) is added to CUM-DWT(X) only if instance D' is FOUND. In the proposed mechanism, whenever D' has FOUNDWT greater than 0., that is, whenever D' is partially found, then some function of DWT(D,X) and FOUNDWT(D') (for example, DWT(D,X) times FOUNDWT(D')) will enter into CUM-DWT(X). If D' is a primitive, then FOUNDWT(D') might be either 0. or 1., as in the current HPL, or it might be a continuous function between 0. and 1., following the suggestions of Section 2.

A disadvantage of the mechanism just proposed is that it could be extremely time-consuming. The FOUND/NOT-FOUND approach was in fact used to avoid just this problem. In the partially found mechanism, CUM-DWT(X) contains factors DWT(D,X) and FOUNDWT(D'). Suppose a descriptor of D' is a primitive which has just been computed, changing CUM-DWT(D') and FOUNDWT(D'). Since FOUNDWT(D') affects CUM-DWT(X), then it must be recomputed also. This changes FOUNDWT(X), which in turn changes the CUM-DWT of all its implicands, and so forth. Whenever the FOUNDWT of an instance changes, all higher-level instances linked to that

instance must have their CUM-DWTs and FOUNDWTs recomputed. HPL could spend most of its time recomputing instance weights, rather than computing primitives! The FOUND/NOT-FOUND approach intentionally avoids this problem. With respect to a particular descriptor D, CUM-DWT(X) is updated only once, and that is when D' is FOUND. It is not updated whenever FOUNDWT(D') changes.

There is, fortunately, a compromise between the two approaches, not as fast as the FOUND/NOT-FOUND approach nor as slow as the approach just described, but maintaining the advantages of the latter. The above mechanism uses a continuous FOUNDWT; the FOUND/NOT-FOUND approach uses, in effect, a discrete FOUNDWT having values 0. or 1. A compromise is to use a discrete FOUNDWT having more than two values. Suppose FOUNDWT has the discrete values 0., .1, .2, ..., 1.0, corresponding to 11 confidence levels. A discrete FOUNDWT could easily be obtained by computing a continuous FOUNDWT in the usual way and truncating its value. Now suppose the FOUNDWT of descriptor D' changes from one discrete value to another. As above, this causes recomputation of CUM-DWT(X) and FOUNDWT(X). However, if FOUNDWT(X) before truncation does not change enough to alter the discrete FOUNDWT(X), the recomputation process stops at this point. Implicands of X would not have CUM-DWTs recomputed, for there is no need. They depend only on the discrete FOUNDWT of X, and that has not changed. With this approach, HPL may decide that HOUSE is present with FOUNDWT .9

rather than .92 as in the continuous partial found mechanism, but this is certainly a minor loss compared to the advantage gained over the FOUND/NOT-FOUND approach.

3.1 Occluded Patterns

The extensions discussed in this section and Section 2 improve HPL's ability to recognize occluded patterns. Figure 8-3 shows a simple example in which a SQUARE is partially hidden behind a RECTANGLE. Using LINE primitives and ignoring compounds for simplicity, SQUARE might have LINE primitives as descriptors representing each of its 4 sides. The lines labelled a and b in the figure would exactly match the corresponding descriptors of SQUARE. Lines c and d would have the correct location and slope but would have the wrong length. However, the mechanism proposed in Section 2 would allow c and d to match the corresponding descriptors of SQUARE with MATCHWTs less than 1. In turn, the FOUNDWT of SQUARE would be less than 1., but probably much above 0. In other words, HPL would recognize SQUARE, but with confidence less than 1.

4. Relations

HPL's major shortcoming as a pattern describer is its inability to handle general relations (including n-ary relations). For example, HPL may recognize TRIANGLE at a certain location and then look for SQUARE at a location relative to that of TRIANGLE,

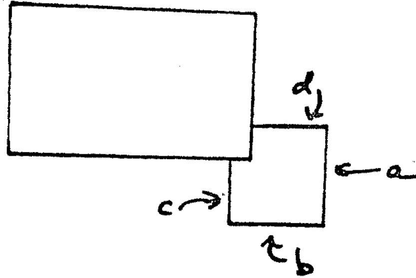


Figure 8-3. An occluded square.

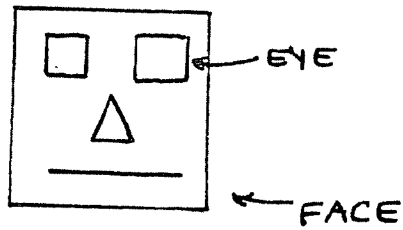


Figure 8-4. A simple illustration of the context problem.

but it will not look for SQUARE anywhere ABOVE TRIANGLE, where ABOVE is a relation named within the memory structure.

HPL does use <u>implicit</u> relations, the relative location and size relations between descriptors, and the subpattern/pattern relations established by descriptors. However, it is difficult to generalize an implicit relation. Therefore, we need the ability to specify an explicit relation as a characterizer, which may then describe other characterizers. The description weight mechanism would represent the importance of a relation. For instance, with an ABOVE characterizer, HPL could describe structures in which it is of great or little importance that one part be above another.

In HPL arguments of a characterizer reference do not depend on the presence of other characterizers. In order to express a relation, some arguments must refer to other characterizers, e.g.

$\overset{1}{[}$TRIANGLE...] $\overset{2}{[}$SQUARE...] [ABOVE #1 #2].

(The number above a characterizer reference identifies it so that the relation may refer to it.) Such arguments will be called "indirect arguments".

Relations can probably be fairly easily added to HPL as primitives. A set of built-in functions such as ABOVE, BELOW, LEFT-OF and RIGHT-OF might be written, for example, [ABOVE x y v], where x and y are indirect arguments pointing to LINE characterizers and v indicates approximately how far x is above y. In the recognition algorithm, investigation of an ABOVE characterizer would require searching for a LINE above a LINE already FOUND, or

determining whether two LINEs already FOUND satisfy the relation.

An argument of a characterizer reference, such as location, might alternatively be expressed as a unary "relation". For example, rather than [LINE LOC...], we could have

1
[LINE...] [LOC #1 v]. Then a set of primitive numerical comparison relations could be defined, such as EQUAL, LESS-THAN, etc., which compare two locations or sizes.

It is interesting to speculate whether <u>non-primitive relations</u> may be learned. Suppose the following characterizers were FOUND:

1 2 3 4
[LINE...] [SLOPE #1 v] [LINE...] [SLOPE #3 v] [EQUAL #2 #4]

and that a sophisticated compound-forming mechanism created compound C1 containing those characterizers. C1 represents the information that LINEs 1 and 3 are parallel. In HPL C1 would not be a relation [C1 x y] but an independent characterizer.

Why is the relation [C1 x y] preferable to the independent characterizer C1? Suppose a higher-level characterizer contains

1 2
[LINE...] [LINE...] [C1] as descriptors. Finding C1 means finding two parallel LINEs, but not necessarily LINEs 1 and 2. The relation [C1 #1 #2], however, directly establishes the correspondence between LINEs 1 and 2 and the descriptors of C1. This correspondence is in fact what makes a relation important.

If HPL can build the non-relational compound C1, then with feedback indicating the relation name and what characterizers are related, a relational goal [PARALLEL x y] could possibly be learned; however, the more difficult problem is to generate

relations without feedback. (For the reader interested in pursuing this problem, Becker's[1970] speculative discussion of "secondary kernel formation" may be useful.)

5. The Context Problem

The context problem is the issue of identifying a pattern which is not uniquely specified by shape alone. Figure 8-4 shows an example, a FACE in which EYE is represented as a SQUARE. We do not want any occurrence of SQUARE to be called EYE, only the specific occurrences in FACE at their proper locations.

5.1 Guzman's Approach to Context

Guzman[1971] proposed an approach to context within a pattern recognition system based on syntax-rules, such as

EYE + EYE + NOSE + ... = FACE

(which we have simplified by ignoring positional information). Suppose a subpattern is recognized which satisfies the syntax rules defining both EYE and WHEEL. Guzman's recognition mechanism, a parsing algorithm, will choose one of the possible names, e.g. EYE. If none of the other parts of the syntax rule defining FACE are present, the algorithm will backtrack and try the other possibility, WHEEL. If no syntax rules containing WHEEL are satisfied, the algorithm will backtrack further.

Our main objection to Guzman's approach concerns the use of syntax rules, which are at present too rigid (as discussed in

Chapter 2). In addition, backtracking does not easily fit into HPL's structure. Once a goal such as EYE is FOUND, it seems unconvincing to decide at a later time in processing that EYE is NOT-FOUND and that WHEEL is instead FOUND. Rather, we prefer an approach in which contextual information enters directly into the decision as to whether EYE is FOUND.

5.2 An Approach to Context within HPL

In HPL, a characterizer is FOUND when its description-set has been satisfied. Our approach to context expands this concept of when a characterizer is FOUND to include its implication-set, the characterizers which it describes. The implication-set is, in effect, the context. In HPL a description weight applies to descriptors and an implication weight applies to implicands, but the two weights are not analogous. IWT is treated as a probability and has a different function during recognition than DWT. We propose a new weight associated with each implicand which _is_ analogous to DWT. This weight will be called a context weight (CXWT), although it might be more consistent to call it an implicand description weight.

Each implicand would have the form:

[(NAME LOC SIZE) IWT _CXWT_].

In the same way that DWTs are added to CUM-DWT for each instance, CXWTs are added to a cumulative context weight (CUM-CXWT). Similarly, in addition to the THRESHOLD, which applies to the

description-set, we introduce a second threshold, CXTHRESH, applying to the implication-set. An instance is FOUND when its CUM-DWT exceeds THRESHOLD. We shall call an instance CONTEXT-FOUND when it is both FOUND and its CUM-CXWT is greater than or equal to CXTHRESH. A goal instance will be output as a response only if it is CONTEXT-FOUND. (If CXTHRESH is 0, a FOUND instance is automatically CONTEXT-FOUND, indicating it does not depend on context. Therefore, the proposed mechanism includes HPL's current mechanism as a special case.)

The most important part of this approach is the distinction between FOUND and CONTEXT-FOUND; the former in effect sends information upward in the memory graph, the latter downward. (Backtracking similarly sends information downward.) When an instance X is FOUND, the CUM-DWT of each <u>implicand</u> A is adjusted by DWT(X,A), an upward flow of information. This may, in turn, cause A to be FOUND. Suppose at some point an instance A is both FOUND and CONTEXT-FOUND (e.g., because CXTHRESH is 0). Then the CUM-CXWT of each <u>descriptor</u> X of A is adjusted by the CXWT of A in the implication-set of X, a downward flow of information. This may, in turn, cause X to be CONTEXT-FOUND, and so forth. To summarize, if an instance is CONTEXT-FOUND, not only has its description-set been satisfied, but its implication-set has been satisfied; the requisite implicands have been CONTEXT-FOUND.

Suppose the description-set of goal EYE has been satisfied, so that EYE is FOUND. EYE is a descriptor of FACE, along with NOSE,

MOUTH, etc. If these descriptors are also FOUND, then FACE will be FOUND. Suppose that FACE does not depend on context (CXTHRESH is 0) and, hence, is CONTEXT-FOUND. Then EYE, NOSE, etc., will also be CONTEXT-FOUND, and given as responses. If the other descriptors were not FOUND, or FACE depended on PERSON as context and PERSON was not CONTEXT-FOUND, then FACE would be FOUND but not CONTEXT-FOUND, and hence not output, nor would EYE, etc.

This approach may appear artificial, but it fits into HPL's structure without great modification. The artificiality is more a result of the FOUND/NOT-FOUND approach than the context mechanism. This mechanism would fit very well with the partially found approach discussed in Section 3. Rather than being FOUND, a characterizer has a FOUNDWT of a certain value. Similarly, rather than being CONTEXT-FOUND, a characterizer has a CXFOUNDWT which depends both on its FOUNDWT and CUM-CXWT / CXTHRESH. Positive FOUNDWTs send information upward in memory; positive CXFOUNDWTs send information downward. The repeated adjustment of CUM-DWTs, FOUNDWTs, CUM-CXWTs, and CXFOUNDWTs is a continuous version of the discrete decision-making required by Guzman's backtracking approach.

As an example, the goal representing FACE might have a non-zero CXTHRESH so that context, the goal PERSON of which FACE is a descriptor, has some effect. Suppose FACE has a CXFOUNDWT of .7. This could mean that the description-set of FACE has been perfectly satisfied, but PERSON has a low CXFOUNDWT, or, alternatively, that

the description-set of FACE has only been partially satisfied but PERSON has been strongly recognized with a high CXFOUNDWT.

This proposed context mechanism allows context to be flexible in the same way the description-set of a particular characterizer is flexible. Various implicands of a characterizer may have differing importances as context, measured by CXWT. The relation of CXTHRESH to the CXWTs determines the overall importance of context in deciding the presence of a particular feature. Important to the viability of this approach is whether the CXWTs and CXTHRESHs can be meaningfully adjusted by a practical learning algorithm.

5.3 Learning Context Weights

We propose the following algorithm for adjusting CXWTs of implicands of goal nodes, which are subject to external feedback. Suppose pattern X is output but is not present in the input. There are two possible reasons for this error: either the description-set of X is inadequate or the implication-set of X does not adequately account for context.

There are two alternatives for learning context. HPL could be given explicit feedback indicating whether the pattern is simply "WRONG" (not described properly) or "WRONG--CONTEXT" (context inadequately accounted for). In the former case, the DWT adjustment mechanism described in Chapter 6 would be used; in the latter case, CXWTs of CONTEXT-FOUND implicands would be

decremented and all other implicands incremented, and CXTHRESH would be incremented, making it less likely that the characterizer will again be CONTEXT-FOUND under similar circumstances. In some situations, it would be obvious to a trainer that a pattern is wrong because of context, for example, when EYE is output but no FACE is present. The situation may not always be so clear-cut, for example, when part of the description-set and part of the context are present.

We prefer an alternative less dependent on the trainer, e.g., requiring "WRONG" feedback but not separate "WRONG--CONTEXT" feedback. DWTs and CXWTs could be adjusted as just described, but at the same time, letting long-term learning determine the relative importance of the description-set versus the implication-set.

The other type of error HPL can exhibit is not responding with the name of a goal which is present. In this case, DWTs would be adjusted if the goal was FOUND, but not CONTEXT-FOUND, by the usual algorithm. CXWTs of CONTEXT-FOUND implicands would be incremented, other CXWTs would be decremented, and CXTHRESH would be decremented. If EYE and its context FACE both occur in the input but EYE is not a response, the CXWT of FACE in the implication-set of EYE would be incremented and CXTHRESH decremented, making it more likely EYE will be CONTEXT-FOUND the next time.

The above proposals for adjusting CXWTs depend on external feedback and apply only to goals. Context weights are not

meaningful for primitives, since they are defined by built-in functions, and cannot send contextual information "downward" in memory (for they are at the "bottom" of memory). We can suggest a CXWT adjusting mechanism for compounds, modifying CXWTs if a compound is FOUND but not CXFOUND, and modifying DWTs if a compound is not FOUND but has its CXTHRESH exceeded. This mechanism, however, is more problematic than the one suggested for goals. Feedback ultimately determines whether a goal is present in the input, in a sense anchoring it to the external world, but a compound is defined internally. Experimentation is needed to determine whether CXWTs for compounds will converge over time to meaningful values, or whether too many parameters to learn have been introduced.

CHAPTER 9

CONCLUSIONS AND FURTHER EXTENSIONS OF HPL

1. Introduction

In this concluding chapter, we compare HPL in detail with Winston's [1970] pattern description program, examining issues of weights and learning, followed by a discussion of further extensions of HPL (curved lines, two-dimensional projections of three-dimensional patterns, and "general to specific" recognition). We then briefly discuss computer language facilities we would have liked to use in designing HPL, followed by a summary of the thesis.

2. A Comparison of HPL with the Work of Winston

Winston's [1970] pattern and scene analysis program (which we shall refer to as WP) invites comparison with HPL, for it uses a net-structure memory and to some extent learns. WP builds net-structure descriptions of scenes composed of two-dimensional projections of three-dimensional polyhedra. Objects are represented as nodes in the net, and relations between objects, such as ABOVE, IN-FRONT-OF, LEFT-OF and SUPPORTED-BY, are represented as labelled arcs between nodes. Relations may also be defined within the net-structure. WP consists of several processes, including description generation, description comparison, model building, and pattern identification.

WP assumes that an input scene has already been processed by a

program such as Guzman's [1968] to segment figures from background and determine the presence of several basic shapes, e.g., WEDGE and BLOCK. Guzman's program is philosophically similar to Winston's, using built-in functions (e.g., a vertex classification routine) to accomplish its goals.

WP generates a description of an input pattern or scene using built-in processes which do not depend on learned information stored in memory. Learned information is only used when WP attempts to identify the description; description is a separate process from identification. Figure 9-1 shows an ARCH and the description WP might generate for it. The name ARCH will not be associated with the description, however, until after the identification process.

Using built-in functions, a net-structure description of a scene may be compared with another net-structure description, an ability which underlies several tasks WP can perform. WP can compare two different scenes by independently generating descriptions of each scene and then comparing those descriptions. WP identifies an input by comparing its description with descriptions of "models" of patterns stored in memory. The result of comparing two net-structure descriptions is, interestingly, another net-structure description, called a "skeleton", which describes how the two descriptions compare.

WP identifies a pattern by comparing its description against stored descriptions of previously learned patterns, called "models". Model-building is the only process in WP which makes use of learning.

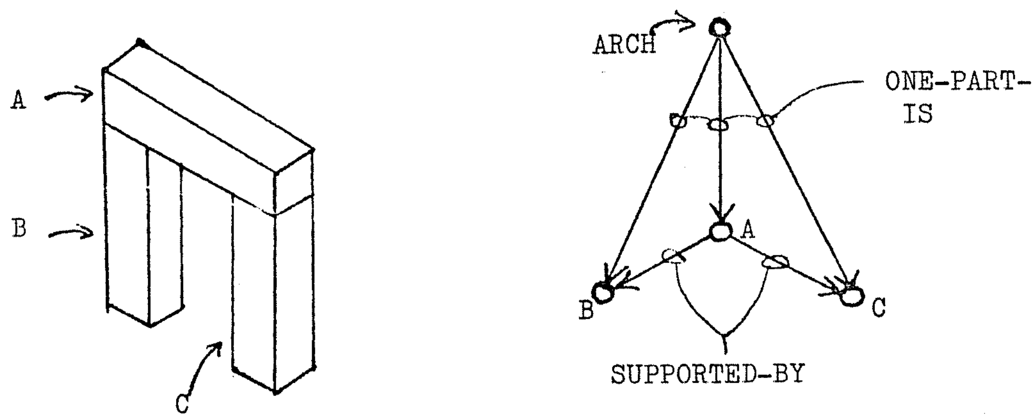


Figure 9-1. An ARCH and its net-structure representation in Winston [1970].

0	1	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
0	1	0	0	0

Figure 9-2. A submatrix that could represent a curve.

Suppose WP has not already learned the pattern ARCH and is presented with an example of it, and its name. A description of the pattern will be generated, the pattern name will be associated with that description, and the description will be saved in memory as a "first-generation" model of the pattern. Suppose another instance of ARCH is now presented, along with its name. Its description is compared with the description of the first-generation model. If they are sufficiently different, a second-generation model may be produced. Continuing this process, a third-generation model may be produced, or a different second-generation model, using a backtracking mechanism. (Learning in WP is further discussed in Section 2.2.)

Pattern identification proceeds as follows. Suppose an example of ARCH is presented. A description of the pattern is generated and is compared with descriptions of models in memory. The models are ranked according to how closely they match the description of the input, and the name associated with the model that best matches is the program's response. Since comparison of two possibly complex descriptions is very time-consuming, comparing the input description with every model in memory is out of the question. To alleviate this problem, models are linked together in memory via "similarity descriptions", a type of comparison skeleton between models. A skeleton is generated between the input description and a particular model. Then that skeleton is compared with similarity descriptions linking the model to similar models, in order to decide which model to try next. Similarity descriptions are intended to make

identification practical, but they, themselves, consume memory and take time to create and process. It is important to know under what circumstances similarity descriptions are formed. Winston merely comments that WP spends "idle time" forming similarity descriptions between models.

Our approach contrasts greatly with Winston's, despite similar goals. We do not believe that object and scene segmentation, description, model building, and identification should be separate processes. The pattern or scene "description" which HPL produces, the set of characterizers FOUND in the input, is developed at the same time as the process of pattern identification and is an integral part of that process. WP identifies by explicit comparison of descriptions; HPL implicitly compares (in matching descriptors to instances) during the course of identification. In Winston's memory, patterns are represented as separate models; they are only tied together through the _ad hoc_ mechanism of similarity descriptions. Winston found it necessary to tie together models in order to direct the search for what model best matches the input description. In HPL's memory, patterns are integrally tied together through characterizers shared in common; this is not _ad hoc_ but inherent in the structure, and it greatly increases efficiency.

WP uses explicit relations in generating descriptions. Relations are built-in, however, and no attempt is made to learn them in a manner analogous to learning patterns. (Explicit relations in HPL are discussed in Chapter 8.)

2.1 Unweighted Links versus Weighted Links in a Net-Structure

All links between nodes in WP's memory are unweighted; two nodes are either linked by a certain relation or they are not. Links in HPL are weighted, ranging from 0, indicating irrelevance, to the maximum DWT (10), indicating the most importance. This gives HPL's memory greater flexibility than an unweighted structure and makes it more amenable to learning. Winston introduced some relations to represent probabilistic information; for example, in building models of patterns, the HAS-PROPERTY-OF relation is expanded into MUST-HAVE-PROPERTY-OF and PROBABLY-HAS-PROPERTY-OF relations. HPL can quite naturally represent probabilistic information within its existing structure.

In addition to the awkwardness of handling probabilistic statements in an unweighted structure, there is another basic problem. In an unweighted structure, discontinuous judgements must frequently be made, both during recognition and memory building; is this link present or not; is this property important or not? Such decisions are bound to be wrong on occasion. Either some tolerance for error or a provision for backtracking must be built into the system. In a weighted structure such as HPL's memory (particularly with partially found characterizers, as discussed in Chapter 8), individual judgements are less important, and there is an inherent tolerance for error. Whether to add a particular characterizer to a description-set, for example, while of some importance from the standpoint of memory utilization, is not crucially important. If it turns out to

be of no value as a descriptor, it will eventually reach a zero weight and be eliminated. Similarly, if a particular characterizer is expected but not found, it will not drastically affect the recognition process.

A proponent of unweighted links might argue that in a weighted system, although any individual decision, e.g, a decision to form a memory link, is not crucial, the overall effect of many such decisions is important. We agree, and hence feel the need for feedback, both internal and external. A large number of bad decisions about what descriptors to add or what characterizers to form may result in a very cumbersome memory, using excessive amounts of storage and requiring a great deal of processing time. Some of the mechanisms described in Chapter 6 are first attempts to handle such problems, notably the use of ITM; certainly, much more work is needed in this regard. However, such mechanisms will be gross mechanisms that handle all characterizers in a similar way, which is a virtue. A similar mechanism in an unweighted structure is more likely to be a specific mechanism, relating, for example, to the formation of some particular memory link.

2.2 Negative Description Weights

WP is taught a pattern during its model building process by means of a series of training sequences. WP is presented a pattern and told that it is either an instance of pattern P or a "near-miss" to P, a pattern chosen by the trainer to resemble P but lacking

some important property or including an erroneous one. In generating a new model of P, WP's heuristics take into account whether a positive instance of P or a near-miss has been presented. A near-miss will typically cause the formation of new links in the model, representing properties that should not be present in the pattern. For illustration, suppose WP is learning the letter "O". (Winston does not discuss the recognition of letters.) As part of the training, a "Q" might be given as a near-miss. The model representing "O" would be expanded by adding a MUST-NOT-HAVE-PROPERTY-OF link to the node representing the short straight line part of "Q".

A MUST-NOT-HAVE-PROPERTY-OF relation is analogous to a high negative description weight in HPL. Negative description weights are not currently used in HPL; however, the recognition process could handle them with little modification. If a descriptor with a negative DWT is FOUND, that DWT will subtract from, rather than add to, the CUM-DWT of the instance. It is, in effect, an inhibiting factor. A descriptor might have a slight or a great inhibiting effect, depending on the value of DWT. If the straight line part of "Q", for example, had a negative DWT equal to the sum of the DWTs of the other descriptors of "O", it would completely inhibit an "O" response.

The main difficulty in using negative description weights is deciding what descriptors with negative weights should be included in the description-set of a characterizer. This is analogous to WP deciding what nodes should be connected by a MUST-NOT-HAVE-PROPERTY-OF

link. Any non-"O" pattern, e.g., "X", has properties which do not pertain to "O". It is absurd to include in the description-set of "O" with negative weights descriptors of "X". More reasonably, negative descriptors of a pattern P should be properties which P does not possess but which are properties of patterns likely to be confused with P.

A trainer giving near-miss feedback must specifically decide what patterns are similar enough to the desired pattern to cause problems; it forces the trainer to decide what the crucial properties of a pattern are. Near-miss feedback could be easily integrated into HPL. Suppose a pattern P' is presented, a near-miss of some pattern P. HPL will build a description of P' in its usual fashion, possibly outputting the name of P as an incorrect response. Without modification, HPL may add new descriptors to the description-set of P', since it is a FOUND goal (as described in Chapter 6), but it will not add new descriptors to P, since P is not FOUND. A simple change would allow specifying to HPL that P' is a near-miss to P. Then new descriptors may be added to the description-set of P, as well, but with negative description weights, since those descriptors describe the near-miss P'. For example, if pattern P is the letter "O" and "Q" is the current input, characterizers representing the short straight-line segment of the "Q" will be added to the description-set of "O" with negative DWTs.

We believe that in a learning program, the program itself, rather than a trainer, should be responsible for deciding what

properties are essential to a pattern. HPL could decide for itself what a near-miss is. Suppose that HPL is presented pattern P' and HPL outputs the wrong response P, causing "NO" feedback to be given. In addition to the learning processes that affect goal P', HPL might also add negative descriptors to the description-set of P, as described above, especially if its FOUNDWT were high enough, indicating that HPL was confused. In other words, HPL could consider a near-miss to P as any pattern which is similar enough to P to produce a confident wrong response. HPL may sometimes add irrelevant negative descriptors, but, over time, they will be eliminated through weight adjustment.

3. Further Extensions of HPL

3.1 Curved Lines

HPL has only recognized straight line drawings. In order to handle patterns containing curved lines, the set of primitives can be extended in the following ways:

Curved line drawings could be handled quite simply with suitable submatrix primitives (e.g., the submatrix shown in Figure 9-2) using a fine enough grid to represent the input. As the number of submatrices increases, however, time and memory requirements also increase. This may, in turn, necessitate using a more efficient implementation or a trimmed-down version of HPL.

Slope is built-into LINE primitives. Curvature could also be

built-in as a discrete quantity. With our idealized patterns, this would only require modifying the special input format.

In short, curved lines do not seem to pose great problems to HPL except in increasing the number of submatrix primitives or adding an attribute to LINE primitives.

3.2 Two-Dimensional Projections of Three-Dimensional Patterns

HPL has not been tested with two-dimensional projections of three-dimensional patterns but probably can handle them with a rich enough set of primitives. A vertex classification primitive might be especially valuable for this purpose. (Guzman [1968] uses vertex classification in recognizing two-dimensional projections of polyhedra.)

HPL could be taught, for example, that the pattern in Figure 9-3(a) is a CUBE. Any pattern sufficiently similar (with respect to the primitives) will also be recognized as a CUBE. Using a vertex classification primitive in addition to the LINE primitives, HPL would also recognize the slightly rotated pattern of Figure 9-3(b) as a CUBE. HPL will not recognize the pattern in Figure 9-3(c) as a CUBE, since its vertex structure is not the same as that of the previous examples of CUBE. HPL must be explicitly taught that this pattern is a different view of a cube, e.g. CUBE1 (analogous to learning TRIANGLE1, TRIANGLE2, etc., as discussed in Chapter 7).

We shall briefly speculate on what may be required for a full

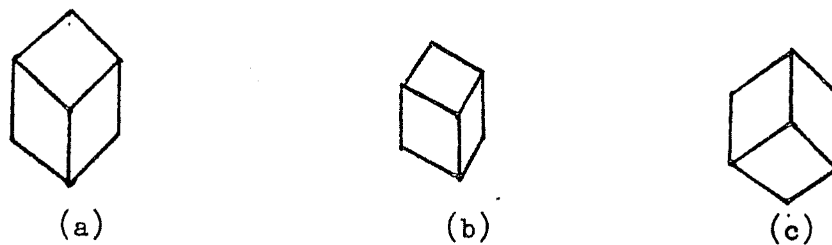
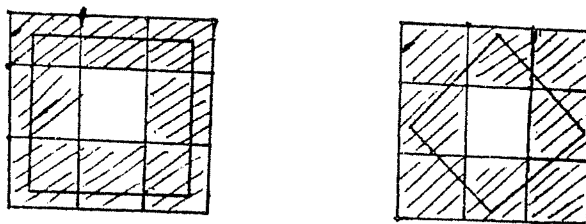
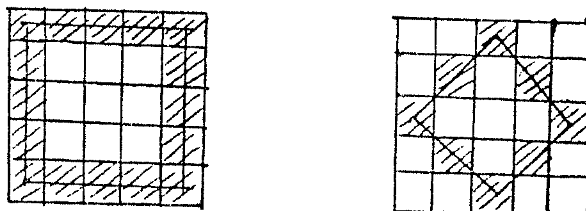


Figure 9-3. Two-dimensional projections of a cube.



A coarse primitive, PC1, with two different patterns



A fine primitive, PF1 A fine primitive, PF2

Figure 9-4. A coarse and fine submatrix with two different patterns.

three-dimensional recognition ability. HPL contains no assumptions about dimensionality except in the subroutines that compute location and size. Those subroutines could be expanded in a straightforward manner to handle three-dimensions. However, primitives must be provided to compute three-dimensional properties. A three-dimensional matrix representation of both the input and primitives can be easily imagined, an extension of the matrix input and primitives already discussed, but how is this input matrix to be obtained? Preprocessing a binocular view consisting of two two-dimensional projections is one possibility.

A more important problem in three-dimensional recognition is the issue of hidden information. In a binocular representation of an input, for instance, much of a pattern is hidden. We have suggested in Chapter 8 how incomplete patterns may be recognized in HPL, particularly allowing for "partially found" patterns; this approach may prove useful in three-dimensional recognition.

3.3 General to Specific Recognition

In human pattern recognition, general features of a pattern may be observed, then more detailed features, and so on. We shall call this "general to specific recognition" and believe it may be more representative of human recognition than the opposite approach, commonly used in mechanical pattern recognizers. For example, a person looking at a forest is not usually overwhelmed by the great detail in the scene; rather, he may observe broad outlines of trees,

then individual trees, then perhaps details of a particular tree.

General to specific recognition may be advantageous in coping with a large amount of input information. The more general the information in a pattern, the less of it there is to process. This general information can then direct recognition to specific details.

General to specific recognition is relevant to the "face in the crowd" problem. We doubt that a person processes in detail each face in a crowd; more likely, general features are observed first. The general features of a certain face may be similar to those of a face described in memory, causing more detailed recognition of that face.

A possible approach to this problem within HPL uses submatrix primitives. An input could be represented not by one matrix, but by two, a fine grid representation (e.g. 20 x 20) and a coarse grid representation (e.g. 8 x 8). Similarly, two sets of submatrix primitives would be used, one set for each grid. A coarse primitive would generally be less costly and more likely to succeed than a fine primitive. Figure 9-4 shows a simple example in which two different patterns are identical with respect to a coarse primitive, but are different with respect to a fine primitive. The coarse primitive, PC1, implies two fine primitives, PF1 and PF2.

Without modification to HPL's structure, both coarse and fine primitives could describe higher-level compounds and goals. Suppose that a compound C1 is described by PC1 and PF1 and that this

compound is a descriptor of some other characterizer currently being investigated by HPL during the recognition process. C1 will be placed on the ACTIVE list, and at some later time may be investigated. When this occurs, its descriptors PC1 and PF1 will be placed on ACTIVE, but PC1 will have a higher AWT than PF1, since it is more probable and less costly. If PC1 is FOUND, PF1 will be given a higher AWT and will be more likely to be computed; if it fails, PF1 will be given a lower AWT. Compound C1 represents that the coarse primitive PC1 "implies" the fine primitive PF1; it directs recognition from a general to a specific primitive. Compound C1 could be built-in, since it represents a fixed relationship between coarse and fine primitives; alternatively, no such compounds need be built-in, letting the learning mechanism form compounds which are especially valuable.

Our approach will not force rigid general to specific recognition, because of the probabilistic nature of the recognition algorithm. Coarse primitives will be given preference over fine ones only to the extent that they are deemed more valuable. This mixed approach seems preferable to a rigid one.

More than two grid sizes could be used in our general to specific approach, but coarse and fine grids are probably sufficient to determine the merit of our proposal.

4. Language Facilities Used in HPL

HPL was written in the "general-purpose" list-processing language, LISP. We tried to design HPL so that its structure was not strongly influenced by LISP; however, the language used may exert a subtle influence on the structure of a program. We shall briefly discuss some computer language facilities we have used or would have liked to use. Many of these facilities are embodied in the new artificial intelligence languages, such as QLISP [Reboh, et al., 1973].

HPL's net-structure memory representation uses the "association list" feature of LISP to associate a characterizer name with its description-set and implication-set. The description-set and implication-set, in turn, reference other characterizer names, effectively forming links between nodes. HPL was not constrained by this approach; however, it is conceivable that a language might allow explicit definition of a net-structure, including nodes and links between nodes, and the association of properties with nodes and links (such as names and weights).

In designing HPL's data structure, we found it helpful to refer to components of list structures by meaningful names, for instance, "description-set" rather than "first element of association list". We defined LISP functions having meaningful names to access appropriate parts of list structures. It would be convenient if a language allowed defining specific data structures and accessing them with programmer defined names; for example, a CHARACTERIZER

might be defined as a data type consisting of the components

[DESCRIPTION-SET, IMPLICATION-SET, PARAMETERS].

The set of instances HPL creates during each frame uses a moderate amount of memory space. At the end of each frame, these instances are discarded; new instances are created in the next frame. Eventually, LISP automatically garbage collects unused memory, including discarded instances, a time-consuming process. It would be more efficient to have some control over the memory allocation process; for example, a separate fixed length memory area could be used solely for storing instances. This area would be reused in each frame.

HPL was tested more extensively with LINE primitives than submatrices, primarily because of the expense of computing submatrices in 1108 LISP. It would have been more efficient to compute primitives in a lower-level language, such as FORTRAN, maintaining the remainder of HPL in LISP. This suggests the desirability of interfacing between different languages, perhaps with some common subroutine calling convention or file storage format, readily suspending operation in one language and switching to the other.

4. Concluding Summary

We have described in detail a running computer program, HPL, which learns to recognize patterns composed of subpatterns. We

presented an overview of the program in Chapter 3, introducing its three main subdivisions, the memory structure, the recognition algorithm, and the learning mechanism. In the subsequent three chapters, we discussed each part in detail.

Chapter 4 discussed the memory structure, a homogeneous, hierarchical net-structure in which weights represent the importance of parts in describing wholes and probabilities between parts and wholes. The differences between permanent memory and short term memory were also discussed.

In Chapter 5, we presented the details of the heuristic-search recognition algorithm, which contains both upward-searching and downward-searching subprocesses. We showed how the algorithm attempts to efficiently search memory and compute primitives, using probabilities, costs, and values of memory nodes.

Chapter 6 explained the function of compound characterizers and discussed the learning techniques of weight adjustment, link formation and deletion, and compound analysis.

In Chapter 7, we presented concrete examples of HPL's tested performance, including complex patterns composed of subpatterns, simple scene analysis, and attending to a requested pattern. We also discussed the three different types of primitives which have been used, letters of the alphabet, matrix templates, and straight lines.

In the two remaining chapters, we have discussed extensions

to HPL which would improve its recognition and descriptive ability, including partially found characterizers, relations, context, and general to specific recognition.

We have been encouraged by our results in running HPL and believe important issues, inluding learning, attention, part/whole structure, relations, efficiency, and context, have been discussed. HPL may serve as a general framework for a better understanding of these issues.

REFERENCES

Becker, Joseph D. (1970). An Information-Processing Model of Intermediate-Level Cognition. Stanford Ph.D. Dissertation. Stanford Artificial Intelligence Memo AI-119, Palo Alto, Calif.

Duda, Richard O. and Hart, Peter E. (1973). Pattern Classification and Scene Analysis. Wiley and Sons, New York.

Evans, Thomas G. (1968). A grammar-controlled pattern analyzer. Proc. IFIP Congress 68, 1592-1598.

__________ (1969a). Descriptive pattern-analysis techniques: potentialities and problems. In [Watanabe, 1969], 147-157.

__________ (1969b). Descriptive pattern analysis techniques. In [Grasselli, 1969], 79-95.

Falk, G. (1971). Scene analysis based on imperfect edge data. Proc. Second Int. Joint Conf. on Artificial Intelligence, 8-16.

Feldman, J. A., Gips, J., Horning, J. J., and Reder, S. (1969). Grammatical complexity and inference. Technical Report CS 125, Computer Science Department, Stanford University, June, 1969.

Fu, K. S. (1968). Sequential Methods in Pattern Recognition and Machine Learning. Academic Press, New York.

Grasselli, A., ed. (1969). Automatic Interpretation and Classification of Images. Academic Press, New York.

Guzman, Adolfo (1968). Decomposition of a visual scene into three-dimensional bodies. Proc. AFIPS 1968 Fall Jt. Comp. Conf., Vol. 33, Thompson Book Co., Washington, D. C., 291-304.

__________ (1971). Analysis of curved line drawings using context and global information. In [Michie, 1971], 325-375.

Londe, D. and Simmons, R. (1965). NAMER: a pattern recognition system for generating sentences about relations between line drawings. Proc. ACM 20th National Conference, 162-175.

McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., and Levin, M. I. (1962). LISP 1.5 Programmer's Manual. MIT Press, Cambridge, Mass.

Michie, D. and Melzer, B., eds. (1971). Machine Intelligence 6. American Elsevier, New York.

________________ (1972). Machine Intelligence 7. American Elsevier, New York.

Miller, W. F. and Shaw, A. C. (1968). Linguistic methods in picture processing--a survey. Proc. AFIPS 1968 Fall Jt. Comp. Conf., Vol. 33, No. 1, 279-290.

Nagy, G. (1968). State of the art in pattern recognition. Proc. IEEE, Vol. 56, 836-862.

Nilsson, Nils J. (1965). Learning Machines. McGraw-Hall, New York.

Reboh, R. and Sacerdoti, E. (1973). A preliminary QLISP manual. SRI Art. Int. Center Technical Note 81, Aug., 1973.

Rosenfeld, A. (1969a). Picture processing by computer. ACM Computing Surveys, Vol. 1, No. 3, 147-176.

________ (1969b). Picture Processing by Computer. Academic Press, New York.

________ (1973). Progress in picture processing: 1969-71. ACM Computing Surveys, Vol. 5, No. 2, 81-108.

Sauvain, R. W. and Uhr, L. (1969). A teachable pattern describing and recognizing program. Pattern Recognition, 1969, 1, 219-232.

Slagle, James R. and Lee, Richard C. T. (1971). Application of game tree searching techniques to sequential pattern recognition. Communications ACM, 14:2, Feb., 1971, 103-110.

Uhr, Leonard, ed. (1966). Pattern Recognition. Wiley and Sons, New York.

________ (1973). Pattern Recognition, Learning, and Thought. Prentice-Hall, New York.

________ and Jordan, Sarah (1969). The learning of parameters for generating compound characterisers for pattern recognition. Proc. 1st Int. Joint Conf. on Artificial Intelligence (Walker, D. E. and Norton, L. M., eds.) Washington, D. C., 381-415.

Uhr, Leonard and Vossler, Charles (1961). A pattern-recognition program that generates, evaluates, and adjusts its own operators. In [Uhr, 1966], 349-364.

Waltz, D. L. (1972). Generating Semantic Descriptions from Drawings of Scenes with Shadows. MIT Ph.D. Dissertation, Art. Int. Lab. Technical Report TR-271, Nov., 1972.

Watanabe, Satosi, ed. (1969). Methodologies of Pattern Recognition. Academic Press, New York.

Winston, Patrick H. (1970). Learning Structural Descriptions from Examples. MIT Ph.D. Dissertation, Project MAC Technical Report TR-76, Sept., 1970.

________________ (1972). The MIT Robot. In [Michie, 1972], 431-463.