A VARIETY OF INTELLIGENT LEARNING IN A GENERAL PROBLEM SOLVER

A. Newell
J. C. Shaw
H. A. Simon*

Mathematics Division
The RAND Corporation

P-1742

July 6, 1959

Presented at the Interdisciplinary Conference on Self-Organizing Systems, The Museum of Science and Industry, Chicago, Illinois, May 5,6, 1959.

*RAND Consultant with Carnegie Institute of Technology

A VARIETY OF INTELLIGENT LEARNING IN A GENERAL PROBLEM SOLVER

The analysis in this paper is part of an exploration of the possibilities for learning and self-organization in a computer program called the General Problem Solver I, or GPS. GPS is a program that incorporates heuristic means for solving a substantial range of problems including, for example, discovering proofs for theorems in logic, proving algebraic and trigonometric identities, and performing formal integration and differentiation. The analysis derives from the following heuristic: To study learning and self-organization, take a program that accomplishes a significant task and discover all the ways it can be improved and can improve itself. Heuristic programs are likely candidates for such an investigation, since, by their very nature, they are open to improvement almost everywhere. We might have chosen for study our chess program or LT, our earlier program for proving theorems in logic, but the reason for preferring GPS will become apparent immediately.

The basic learning situation is depicted in Figure 1. The performance program at the bottom of the figure is GPS. A learning situation requires another program, called the learning program, that operates on the performance program as its object to produce a new performance program better adapted to its task. For GPS, the changes must make it a better problem solver. We will not try to define in general the

notion of adaptive change and its measurement. In each particular learning situation we must convince ourselves that the learning program is so structured that it will try to produce better programs, although it may not always succeed, even in the long run.

Basic Learning Situation

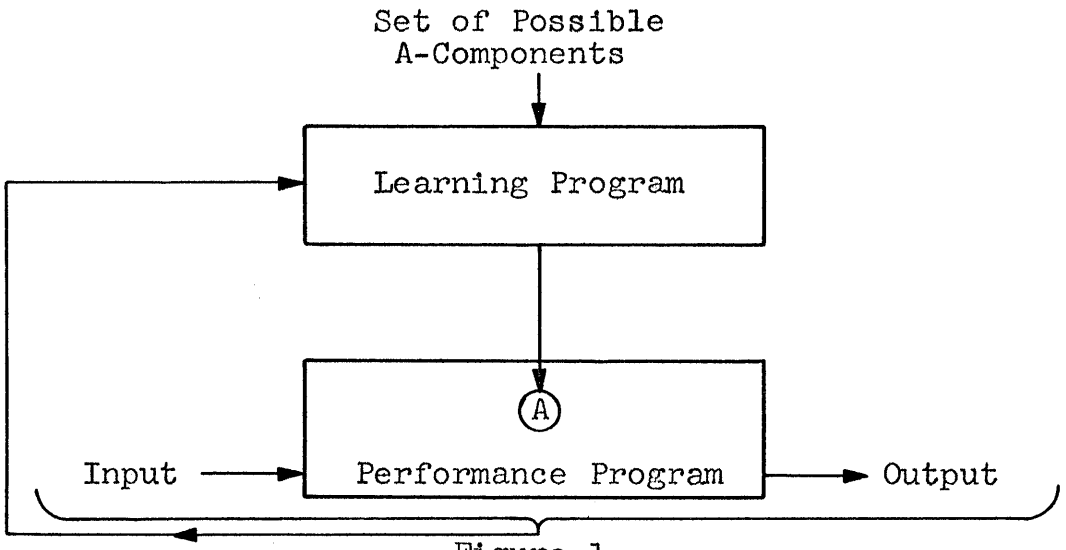


Figure 1

A performance program like GPS is large and complex, and is organized from a set of components or aspects. There will be many different learning opportunities corresponding to the different aspects — it being our ultimate goal to discover each of these and determine its nature. The learning program for any particular aspect, A, must have access to a set of possible components for the performance program. The set may be given simply by a list, or by the variation of numerical parameters. If the aspect selected for learning is

a significant part of the performance system, then the space of possible components will be large and complex. It might consist, for example, of all programs that can be built up from a set of primitive processes. The learning program also must have access to information about the performance program, its inputs, and its outputs. What information is used will vary, of course, but we shall assume that the learning program has essentially complete information about the structure of the performance program and its behavior for a sample of tasks. The learning program may work iteratively over time, selecting candidate A-components, modifying the performance program accordingly, watching the modified program operate, and then repeating the cycle. It need not proceed in this way, however. Although a learning program is constrained, by definition, to produce new and hopefully better programs, it is not constrained to do it in any particular way.

Our problem, then, is to construct a program, which we call a learning program, that will make a good selection of an element from the set of A's, so as to yield an effective performance program — in the present instance, a program that can solve problems. If the performance program handles a significant task, if the A-component chosen is a significant aspect of the performance program, and if the space of A-components is sufficiently rich; construction of the learning

program will pose an interesting problem.*

In designing learning programs we are using a particular heuristic:

> Generally: That significant learning situations will require learning programs that are heuristic problem-solving programs, in the sense in which that word is currently used in discussing chess and theorem-proving programs.**
>
> More specifically: That because GPS has pretensions of solving a wide array of problems, it may be possible to let GPS be its own learning program, so that the problem of selecting an A-component will be a problem of the form GPS can work on.

The purpose of this paper is to follow this heuristic in order to see where it leads. It will become apparent that our analysis is still incomplete, although we have tried to be as definite as we can. Perhaps, even so, we have traced enough of the path so that the reader can evaluate the potentialities and difficulties of this approach.

The General Problem Solver

GPS and its performance have been described in detail in

*Thus the reason some early attempts, like Oettinger's program for conditioned response [6], have not led very far, although they clearly are learning programs, is that the space of components is too simple and regular. Conversely, some of the interest of Friedberg's learning program [2] stems from the fact that the space of components for his program consists of all possible programs — a very large and irregular space.

**Heuristic programs are still best described by example. See [3,5].

other publications [4,5]. We will include here only enough description to make GPS comprehensible to readers who are not already familiar with it.

GPS is a program for working on tasks in an environment consisting of objects and operators. Symbolic logic is one particular task environment in which GPS can operate. Figure 2 shows an example of a simple task in this environment. The objects on the left-hand side of the figure are symbolic logic expressions, or propositions; the operators, on the right-hand side, are the rules that define the admissible transformations of one expression into another. For example, the object, L1, is transformed into the object, L2, by applying the operator, R1. This particular operator is applied by substituting S for A and $(\sim P \supset Q)$ for B in its "input" side, and extracting the corresponding expression (L2) from its "output" side. GPS can solve problems like "Transform L1 into L4." Figure 2 shows a solution for this problem.

Symbolic Logic Problem

OBJECTS		OPERATORS	
L1:	$S.(\sim P \supset Q)$		
L2:	$(\sim P \supset Q).S$	R1:	$A.B \rightarrow B.A$
L3:	$(P \vee Q).S$	R6:	$\sim A \supset B \rightarrow A \vee B$
L4:	$(Q \vee P).S$	R1:	$A \vee B \rightarrow B \vee A$

Figure 2

The objects to which GPS is applied need not be logic expressions, nor the operators rules of logic. Figure 3 depicts schematically the general nature of the GPS task environment. Here the objects are geometric shapes, and the arrows show the possible transformations of one shape into another by application of operators. In this environment the problem might be posed of transforming the three shapes on the left end of the figure into the shape at the far right. GPS should be able to operate on any environment where there are "things" that can be transformed or combined into other things by applying identifiable operators or rules, and where the things are describable — i.e., have features. The significance of this last qualification will become evident in a moment, as we explain how GPS operates.

The principal components of GPS are a set of goal types: Goal types are used to state problems for GPS and are the major units for organizing the problem-solving process. For our present purposes we need to consider only three types of goals: to transform one object, a, into another object, b; to apply an operator, q, to an object, a; and to reduce a difference, d, on an object, a. At the outset, GPS is given a particular goal (e.g., the transform goal in Figure 2, of

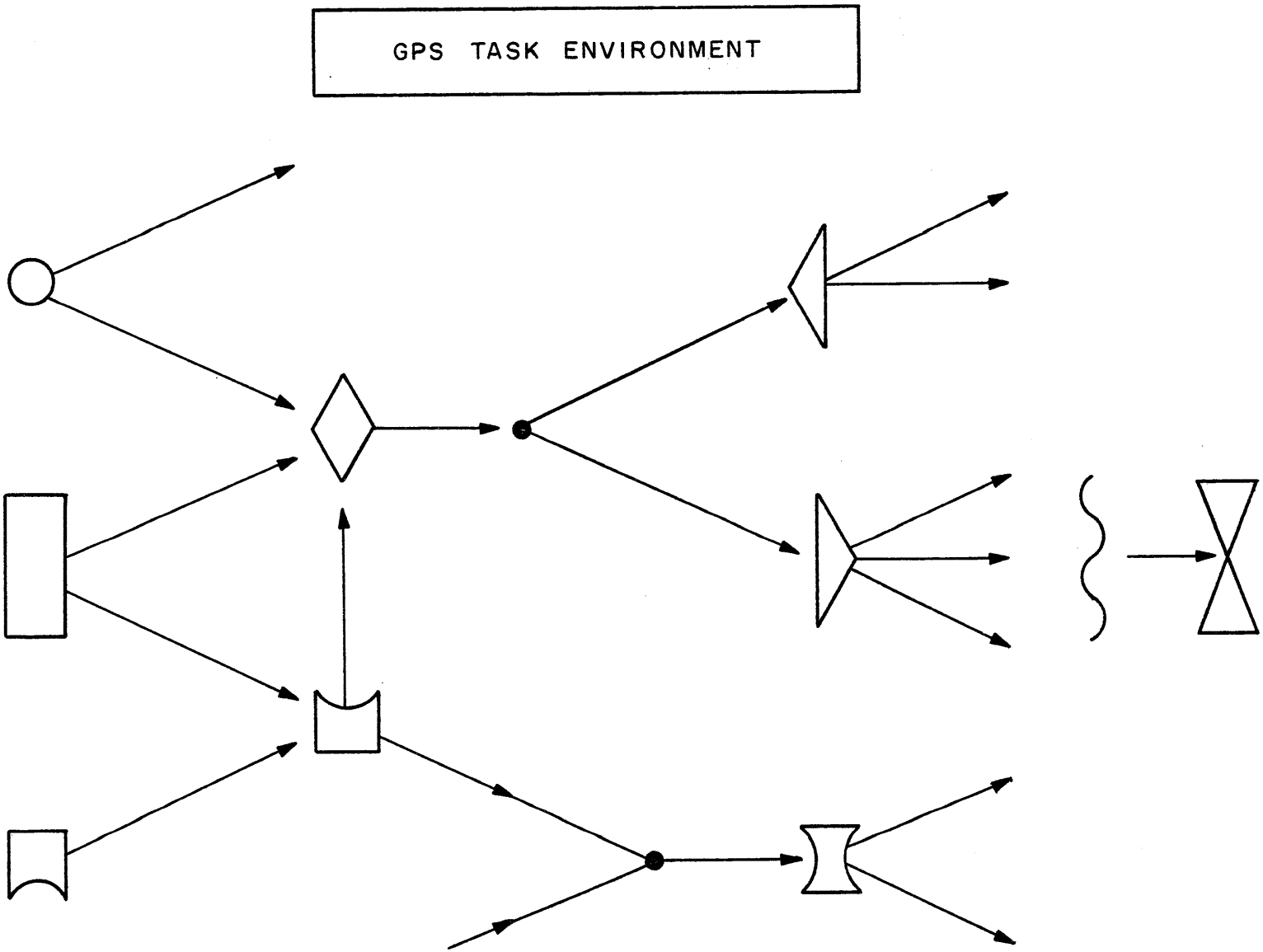


Fig. 3

P-1742
7-6-59
7

changing L1 into L4). It proceeds as follows (Figure 4): It evaluates the goal to see whether it should be worked on; if it accepts the goal, it selects a method associated with that goal, and applies it; if the method fails, it evaluates whether to continue attempting the goal. If so, it selects another method, and so on.

GPS BASIC ORGANIZATION

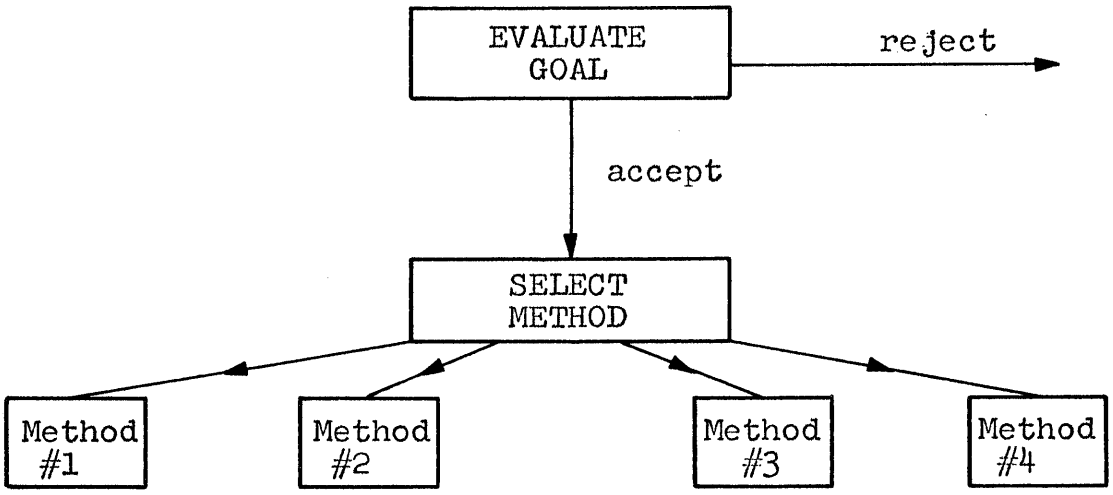


Figure 4

The main clues to the behavior of GPS lie in the methods themselves. These give GPS a basically recursive structure: methods operate by establishing subgoals (belonging to one or another of the three goal types) that are (hopefully) easier than the original goal, until a stage is reached where a subgoal can actually be achieved. When this happens, it represents a step of progress in the goal next above, which can then make progress for the goal above it, and so on.

Figure 5 shows a basic system of methods for what we usually call means-end analysis. It depicts only a core system of interlinking methods used in GPS. Other methods are known but will not be discussed here. The method associated with a transform goal (Type #1) consists in matching the two objects, a and b; discovering a difference, d, between them (if there is no difference, the problem has been solved); establishing the Type #3 goal of reducing d in a; if this is accomplished, producing c from a, establishing the new transform goal of changing c into b. If this last goal is achieved, the original Type #1 goal is achieved.

The method associated with a reduce goal (Type #3) consists in searching for an operator, q, that is relevant to the difference, d; if one is found, setting up the Type #2 goal of applying the operator.

The method associated with an apply goal (Type #2) consists in determining if the operator can be applied by setting up a Type #1 goal for transforming a into an object, C(q), that satisfies the conditions for an input to the operator q. (Generally, an operator can be applied only to objects having certain characteristics. For example, R2 in Figure 2, can be applied only to a logic expression that has a horseshoe ($\supset$) as its main connector.) If this is successful, the operator q is applied to C(q), producing a new object, P(q).

The recursive structure of the program is apparent. Transform goals generate Reduce goals and new Transform goals; Reduce

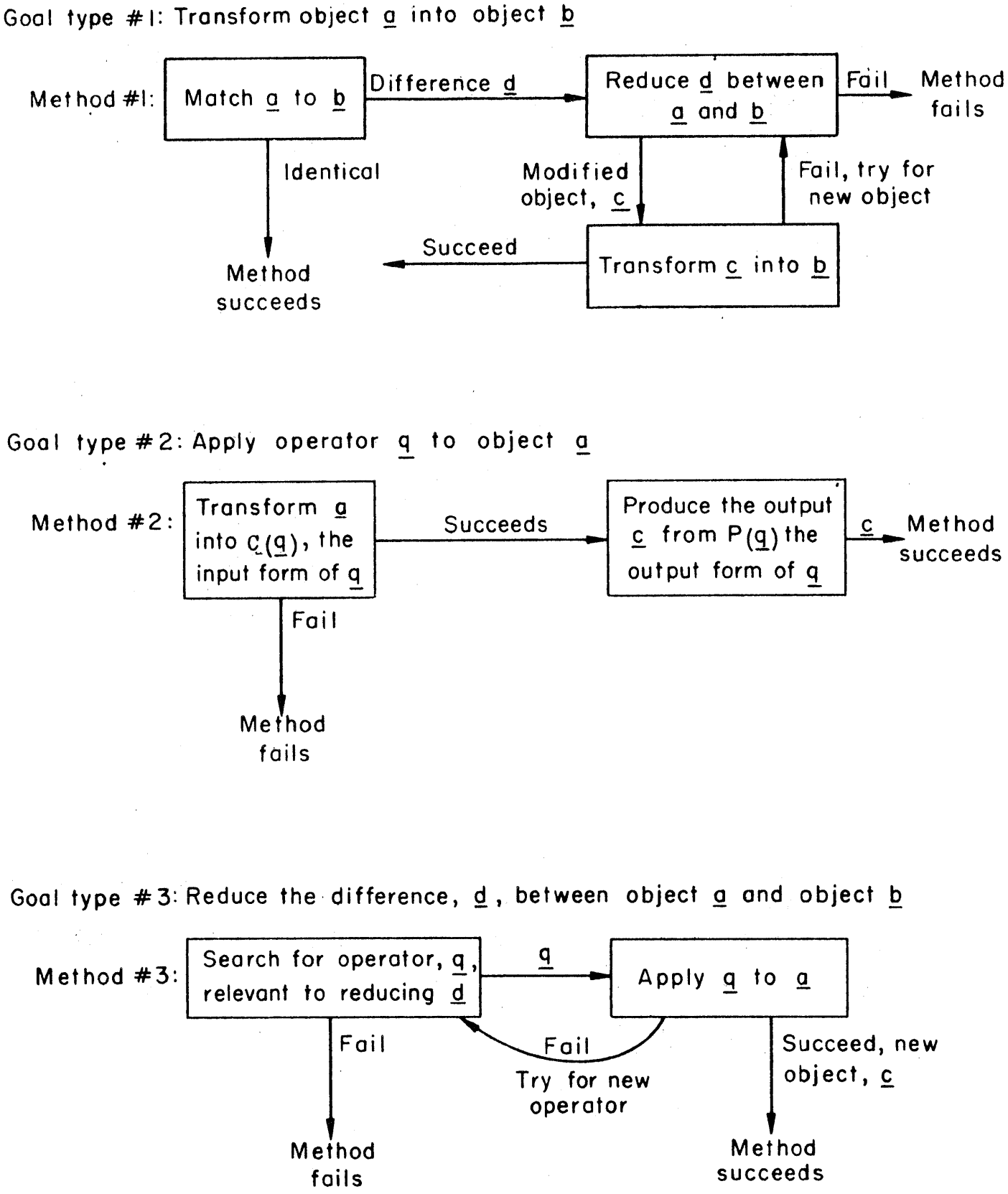


Fig. 5 — Methods for means—ends analysis

goals generate Apply goals; Apply goals generate Transform goals and Apply goals. The additional methods that are known for GPS fall within this same general structure.

We have not yet defined the differences that are part of the definition of the Reduce goals, nor the notion of relevant operators. Specific differences and the tests that detect them are not part of the GPS proper but are parts of each particular task environment to which the program is applied. The stub of the table in Figure 6 is a list of differences that may be detected among logic expressions; a different list might be used for trigonometry, and so on. The columns in Figure 6 correspond to the operators, also specific to the task, in the logic task environment. The x's in the columns indicate which operators, or rules, are relevant to which differences. (How these are determined will be explained later.)

For example, in Figure 2, the goal of transforming L1 into L4 may lead to detection of a difference in position between the two expressions. The operator relevant to this difference (Figure 6) is R1. Thus, the goal of reducing this difference will generate the goal of applying R1 to L1. Since L1 matches the input to R1, the goal will be achieved, producing L2 and generating the new goal of transforming L2 into L4. Using Figure 6 and Figure 2, the reader can simulate GPS's program in carrying through the rest of the solution of this particular simple problem.

The methods just described are ways of setting up subgoals. In the evaluation part of each goal are tests that allow subgoals to be rejected as unprofitable, or to be delayed until after more profitable goals are tried.

LOGIC TABLE OF CONNECTIONS

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
Add variables									X	X		X
Delete variables								X			X	X
Increase number			X				X		X	X		X
Decrease number			X				X				X	X
Change connective					X	X	X					
Change sign		X			X	X						
Change grouping				X			X					
Change position	X	X										

Figure 6

This is all we shall have to say about the performance program itself. We refer the reader to the other publications already cited, in which we show that GPS will, in fact, solve problems in logic, trigonometry, and elementary algebra, and that certain variants of GPS will simulate in considerable detail the behavior of humans performing the same tasks.

The Learning Problem

When GPS is solving problems in a particular task environment, the performance program consists of two parts: (1) GPS proper, the goal types and methods, which are completely independent of subject matter and are not modified in any way when GPS is applied to a new task environment; and (2) the specifications of the particular task environment: its objects, its operators, and its differences. Within a specified environment, of course, there are many different problems — proving all logic theorems or all trigonometric identities, for example.

Learning programs might be devised for either main part of GPS. In the present paper we shall only consider programs to enable GPS to improve its performance in a given task environment — to learn heuristics appropriate to that environment. The learning programs themselves will be general — they are programs for learning about any new environment to which GPS might be applied; the content of what is learned, however, will be specific to a particular task environment. These learning programs do not change the core of GPS; instead,

they modify the specification of the environment, thus making GPS more efficient in solving problems in that environment.

Learning to characterize in an effective way the task environment is an important and prevalent kind of human learning. A problem solver who is experienced in a particular environment will notice features that will be unnoticed by (or even invisible to) an inexperienced person. The native tracker in the forest is a classical example.

The differences listed in Figure 6 are features of the logic task environment that are effective for problem solving in that environment. A problem solver who does not have available a good set of differences has little means for working toward his goal except to try, at random or by rote, different sequences of operators until he gets the answer. He cannot even measure progress easily, for the most direct clue to progress is the elimination of differences between the terminal expression and the expressions he has obtained.

After the problem solver has learned to recognize and attend to a useful set of differences, other things remain to be learned about the environment. If a particular difference appears, what operator shall he apply to remove it? He might search the list of operators, again randomly or systematically, until he found one that affected the difference to which he attended. A more efficient procedure would be to build up, once and for all, the table of connections depicted in Figure 6, which indicates which operators are relevant to the removal of

which differences. Equipped with this table, he could, when faced with a difference, consider only those operators from the entire list that are relevant to removing this difference.

It would be possible, in instructing a learner about a new subject, to teach him specifically what differences to attend to and what operators are relevant to what differences: to give him explicitly a list of differences and a table of connections. In teaching humans we seldom do this. We characterize the task environment by more or less adequate descriptions of the objects and operators (the rules of the game), and perhaps guide somewhat his experiences with the environment. We usually leave it up to the learner to acquire the differences and connections inductively. We assume that humans are equipped with learning programs for improving their performance programs in these two respects. The learner is supposed to be able, himself, to develop a theory about the significant characteristics and structure of the task environment, and to incorporate that theory in his problem-solving program.

In the remainder of this paper we shall discuss in some detail learning programs for the two aspects we have just been considering. The first learning problem we shall pose is: Given the operators and the differences in a task environment, to find a good table of connections associating relevant operators with the several differences. The second problem we shall pose is: Given the objects and operators in a task environment, and a set of basic tests for discriminating features of objects, to find a good set of differences for

that environment. The answers to both questions will take the form of proposed learning programs.

These two aspects hardly exhaust the possibilities for GPS to learn about the particular task environment that confronts it. For example, GPS might learn new operators other than the set given it initially. It might also learn special methods that apply to subclasses of problems. Or it might learn cheap tests to indicate when operators are feasible, thus short-circuiting some of the elaborate general machinery.

Learning the Table of Connections

The learning program required to build a table of connections is quite simple. We describe it because it illustrates a fundamental point about such programs. A simple bit of arithmetic performed on the matrix of Figure 6 shows that the number of possible tables of connections is not small. There are eight differences and twelve rules in the logic environment, and since each rule might be relevant to each difference, there are 12x8=96 possible connections.

One might use a simple trial-and-error learning scheme to build up the table. By simple trial and error we mean a scheme with the following general characteristics: (1) Begin with an arbitrary table (perhaps the one that includes all connections). (2) Keep statistics on how often each rule serves to reduce each difference. (3) Try the several rules with frequencies proportional to their relative successes. This procedure incorporates the simplest kind of mechanism of natural selection,

to use evolutionary language, or reinforcement of correct responses, to use psychological language.

Most learning programs that have been proposed for computers have this simple character. In some general sense, such learning programs will presumably "work." What is not clear is whether they will work within reasonable time limits in environments of the size and complexity of those we encounter in problem solving.

Such a mechanism might work in the case before us, since the set of possibilities seems fairly regular and small. However, it is not evident that it would be efficient. More important, there is no reason why we should limit ourselves to such mechanisms, which operate entirely inductively from performance, when other information is available. In this case there is information about the structure of the operators which can be used to construct the table of connections directly, without a tedious inductive search.

In the logic environment, each operator is given as a form. Rule 1, for example, will accept as input any expression of the form (A.B) and produce as output an expression of the form (B.A). Now by applying, in turn, the tests for each of the differences to the pair of objects consisting of the input and output form of the rule — that is, to (A.B) and (B.A) — it will be apparent that the only difference between output and input is in the position of the terms. It becomes equally apparent that if Rule 1 operates on another expression,

after the latter has been matched to the input form of the rule, it will change only the position of terms in the expression. Hence, the only difference to which Rule 1 is relevant is a difference in position, and the only entry we make in the first column of the table of connections is in the last row.

The remainder of the table can be constructed in the same way. The learning program consists simply in this: Consider each operator in turn. Apply, successively, each test for a difference to the operator. If the result of the test is positive, record the operator in question on the list of operators relevant to the given difference. The list of such lists, over the whole set of differences, is precisely the table of connections.

Let us summarize the analysis briefly. The problem of this particular learning program is to select an element from a set (the set of all possible tables of connections) that satisfies certain conditions. How the learning program can solve this problem depends on what information is available to it. If the program can discover only how the performance program behaves when a given element from the set is incorporated in it, then the learning program can do little more than search blindly and select the elements that work well. If other information is available, however, as in our example, the learning program can incorporate other processes which may be far more efficient than simple trial and error mechanisms. The important empirical question is this: When we consider the

learning problems that arise naturally in complex intelligent systems, what is the nature of the information that is available? Is it so scanty as to restrict learning to simple natural selection, or does it allow other, more sophisticated schemes?

With this basic question in mind, we can now examine the second, more complex, learning situation.

Learning a Set of Differences

The second learning situation may be described thus: Given the objects and operators in a task environment, and a set of basic processes for detecting and discriminating features of objects, to find a good set of differences between pairs of objects for GPS in that environment.

When we try to define a set of possible differences we discover why this learning task is both difficult and interesting. By a "difference" between two objects we mean, of course, some characteristic by which they can be distinguished. For the performance program to make use of differences, these must be incorporated in the program in the form of tests that make the appropriate discriminations. For example, for the program to detect that two logic expressions have different connectives, there must be a test that compares the two connectives and records them as identical or different. These tests are, of course, subroutines in the performance program, and the learning program must be capable of constructing such subroutines — of writing at least a specialized class of programs.

Unlike the earlier situation with the table of connections,

we are here not given the set of possible aspects in any natural way. Before we can discuss the learning program for differences, we must define a programming language from which difference programs can be generated. The programming language must be rich enough to allow adequate learning potentialities; yet simple enough so that the learning program can construct viable routines. And it must not be simply a list of differences already provided for the learner to try. We turn now to the construction of such a programming language.

Programming Language for Differences. A program to test for a difference must be built up out of some set of more elementary processes that are assumed already available to the learning program for assembly. Prominent among these processes will be a set of primitive discriminations. In some sense they must be more elementary than the differences eventually needed, or the suspicion will remain that the important learning occurred in the selection of these primitive notions and not in the assembly of a program from them. Two devices are available to avoid circularity. We can use a very small set of primitive notions; and we can insist that these same notions be applicable to more than a single task environment. For example, we might start with a computer machine code, and require that all differences be programmed in it for all environments.

We have tried to combine both of these criteria in a Difference Program Language (DPL, for the purposes of this paper). DPL has two parts: There is a general part that

consists of a small number of processes for operating on sets and lists. This part is to be used for all task environments, and contains no information about the nature of particular task environments. The second part consists of a list of processes particular to each task environment. These processes constitute basic manipulations and perceptions about the objects in the task environment and form the totality of information about the environment. This list must be given de novo for each environment, since we do not assume that there is a common field (as, in perception, the visual field) in which all objects from the several environments are presented. That is, we proceed as if GPS had a different "sense modality" for each environment, and hence must abstract from this into representations of objects belonging to the general part of DPL before it can describe these objects in terms of common properties or conventions.

Figure 7 gives a list of processes for the environment of symbolic logic. The symbol, $\emptyset$, which represents the null set, is used to record that a test was negative or that a find process had a null output. Note that none of the processes of Figure 7 can be omitted (without some equivalent replacement) if the set is to be complete for logic expressions. If one or more processes were deleted from the list, and the remaining processes were our only source of information about features of logic expressions, we could never become aware of the missing elements. Each process accepts only certain types

PROCESSES FOR SYMBOLIC LOGIC ENVIRONMENT

Symbol	Name	Input	Output	Example
t	Find terms	object	set of all sub-objects	$t(P \vee Q) = \{P \vee Q, P, Q\}$
l	Find left	object	left hand sub-object (ϕ if doesn't exist)	$l(P \vee Q) = P$, $l(Q) = \phi$
r	Find right	object	right hand sub-object (ϕ if doesn't exist)	$r(P \vee -Q) = -Q$
c	Find connective	object	main connective of object (ϕ if object is variable)	$c((P \supset Q) \vee R) = \vee$
s	Find sign	object	main sign of object	$s(-(R.P)) = -$, $s(-R \vee P) = +$
v	Find variable	object	variable letter (ϕ if a compound object)	$v(P \supset Q) = \phi$, $v(-Q) = Q$
b	Test if constant	anything	input, if constant (ϕ if it contains free variables	$b(P \vee Q) = P \vee Q$, $b(A) = \phi$
f	Test if free variable	anything	input, if a free variable (ϕ if not)	$f(A) = A$, $f(P) = \phi$
$\supset$	Test if $\supset$	connective	$\supset$ if connective is $\supset$ (ϕ if not)	$\supset(\supset) = \supset$, $\supset(.) = \phi$
v	Test if v	connective	v if connective is v (ϕ if not)	$\vee(\vee) = \vee$, $\vee(\supset) = \phi$
.	Test if .	connective	. if connective is . (ϕ if not)	$.(.) = .$, $.(\vee) = \phi$
+	Test if +	sign	+ if sign is + (ϕ if not)	$+(+) = +$, $+(-) = \phi$
-	Test if -	sign	- if sign is - (ϕ if not)	$-(-) = -$, $-(+) = \phi$
A (B,C,..)	Test if A (B,C,..)	variable	A if letter is A (ϕ if not)	$A(A) = A$, $A(P) = \phi$
P (Q,R,..)	Test if P (Q,R,..)	variable	P if letter is P (ϕ if not)	$P(P) = P$, $P(Q) = \phi$

Figure 7

of inputs and produces specified types of outputs, as shown in the figure. Besides the list of processes, a list of input and output types is provided. Thus c, the process that finds connectives, can be applied only to objects (logic expressions), and produces a connective, which is a symbol of a different type.*

The general part of DPL consists of the seventeen processes shown in Figure 8. The inputs and outputs of these processes are lists and sets (unordered lists) of items. With two exceptions, the processes treat objects of the task environment as unanalysable units. Thus, the general part of the language is independent of information about particular task environments. One of the exceptions is the process B[X], whose operand may be an object, set or list. This process replaces each component of a specified kind in the input by the null symbol, ϕ. For example B[c] replaces all connectives in an object by ϕ, so that the latter symbol now serves as a generalized abstract connective. Thus B[X] must be able to "get at" all the sub-objects of the object on which it operates.

The other process that requires information about the structure of task environment objects is D, a process that finds differences between pairs of objects. Since this process is of central importance, it deserves extended discussion. The input to D is a pair — that is, a list of two items, say X and Y.

*This information is implicit in the operation of the processes, and could be learned by GPS by a program not very different from that for learning the table of connections.

GENERAL DPL PROCESSES

Symbol	Name	Input	Output	Examples
A[X]	Assign	any object	X if input is not ϕ, ϕ if input is ϕ.	A[+](P,Q) = +, A[+]ϕ = ϕ
$\overline{A}$[X]	Inverse assign	any object	X if input is ϕ, ϕ if input not ϕ.	$\overline{A}$[+](P,Q) = ϕ, A[+]ϕ = +
B[X]	Blank	any object	Goes through all subparts of input: x If X[x] not ϕ, replace X[x] by ϕ in input.	B[c](P$\supset$(QvR) = Pϕ(QϕR)
C	Find component	set	Takes any component for output.	C{$\supset$, v, .} = v
D	Difference	any pair of objects	Compares corresponding subparts of the two inputs. If equal, replaces each by ϕ. Output is modified pair.	D((P,Q,R,),(Q,Q,R,P)) = ((P,ϕ,ϕ), (Q,ϕ,ϕ,P))
E	Expand	set of sets	Output is set of all elements in the subsets of input, with multiplicity.	E{{P,Q}, {P,R}} = {P,Q,P,R}
F	Find first	list	First item on the list (ϕ if doesn't exist).	F(P,Q,R) = P
G[X]	Group	set	Output is a set of sets. Each subset contains all the items of the input set with the same value of X(x).	G[I] {P,P,Q,P,R,Q} = {{P,P,P}, {Q,Q}, {R}}
I	Identity	any object	Input.	IX = X
K[X]	Constant	any object	X, for any input.	K[+]X = +
L	Find last	list	Last item on the list.	L(P,Q,R) = R
M	Find prior	item from list	Item preceding input item (ϕ if doesn't exist).	MP from (Q,R,P,S) = R
N	Find next	item from list	Item following input item (ϕ if doesn't exist).	NP from (Q,R,P,S) = S
P	Intersection	set of sets	Set of items common to all subsets of input.	P{{P,Q}, {Q,R}, {P,Q,R}} = {Q}
R[X]	Select representative	set	Set consisting of one representative element of each value of X(x). Compare G[X].	R[I] {P,P,Q,P,R,Q} = {P,Q,R}
S[X]	Select	set	Set of items of input set with X(x) not ϕ.	S[v] {P.Q, -R, R$\supset$P, R} = {-R,R}
U	Set	list	Set of items on list	U (P,Q,R) = {P,Q,R}

P-1742
7-6-59
214

Figure 8

The members of the pair (X,Y) may be objects or they may be sets, lists, or list structures. The output of D is also a pair of items, say (X', Y'), obtained as follows:

(1) X and Y are put into correspondence according to their structure. For example, if X and Y are logic expressions, they are lined up with their main expressions together, left-hand subexpressions together, right-hand subexpressions together, and so on.

(2) Any corresponding parts of X and Y, respectively, that are identical are replaced, in both X and Y, by ϕ. When this process has been completed for all pairs of parts of X and Y, a new pair of objects, (X',Y') will have been obtained in place of (X,Y). The new pair, (X'Y') is the output of the process D. In this new pair, X' consists of all parts of X not belonging to Y, and Y' of all parts of Y not belonging to X.

The processes are the elementary terms of DPL; we must also provide ways for compounding programs of them. Speaking roughly, DPL programs are sequences of DPL processes. Sequences of processes, each term operating on the output of the preceding ones, are written horizontally. The operation proceeds from right to left as in standard mathematical operator notation. Apart from simple sequences, four other combining operations are needed. These are indicated in Figure 9. For example, the notation permits us to write down a set of processes as though it were a simple process. The output of this set is the set of outputs that would be produced by each of the component

RULES OF COMBINATION

Let P and Q be processes and X and Y be operands.

P.X or PX	=	apply P to X
P* {X,Y}	=	{PX, PY}
P* (X,Y)	=	(PX, PY)
(P,Q):(X,Y)	=	(PX, QY)
{P,Q} X	=	{PX, QX}
(P,Q) X	=	(PX, QX)

Figure 9

processes, operating independently on the input. These various modes of combination of processes are the functional equivalent, in DPL, of such features normally found in programming languages, as iterations, conditional transfers, and working storages.

DPL consists, then, of a set of basic processes and some ways of combining processes. It is not a complete programming language; it omits several important notions, such as ordering relations and recursive definition, in the interest of simplicity. However, DPL is adequate for constructing a rather large set of differences, including those we have used in the performance program of GPS for symbolic logic.

Figure 10 shows, by a step-by-step analysis of an example, how the DPL program will find the difference between the sets of variables contained in two logic expressions. The input

DIFFERENCE IN SET OF VARIABLES

D(R[I]vt)*	(X,Y)
	(P.Q , Q⊃Q)
t*	{P, P.Q, Q}, {Q, Q⊃Q, Q}
v*	{P,Q}, {Q,Q}
R[I]*	{P,Q}, {Q}
D	{P}, {∅}

Figure 10

(X,Y), to the program is the pair of logic expressions, (P.Q, Q⊃Q). The parenthesis followed by the asterisk (*) indicates that the whole sequence of operations, R[I]vt is to be applied to each member of the pair, that is, to X and to Y separately. Then, the differencing operation, D, is to be applied to the resulting pair of outputs. First, application of t to each logic expression produces, for each, a set whose elements are the subobjects included in the original expression (including the expression itself). In our example, each set produced by t contains three elements. Next, application of v to each of these sets replaces each of its elements that

is compound – that is not a variable – by ϕ, and retains the variables. Next, application of R[I] eliminates all multiple occurrences of identical symbols – e.g., it reduces (Q,Q) to (Q). Finally, application of the difference, D, reduces the left-hand set to P and the right-hand set to ϕ. This means, as it should, that the left expression in the original pair contains the variable P, which does not occur in the right expression, but that all variables in the right expression occur in the left.

The most striking characteristic of differences written in DPL is that they are "abstractive." They start with comparisons of the full detail, and gradually remove distinctions by successive application of DPL operations.* This is to be contrasted with normal programming tests, which are "discriminative," in that each elementary process only discriminates a very minute part of the object, and more and more information is built up about the objects by constructing a tree of tests.

Learning Situation for Differences. Having described a relatively rich programming language having a simple structure, which expresses easily some kinds of differences we know to be useful, we can now return to the problem of how a set of differences appropriate to a particular task environment

*DPL is similar in this respect to the language constructed by Selfridge and Dinneen [7,1] for a pattern recognition program. Using the alternative approach Mr. Edward Feigenbaum of Carnegie Institute of Technology, in a forthcoming report, describes a system of discriminative tests that comprise part of a program for simulating human rote learning.

might be learned. The problem can be reformulated thus: Given the objects, operators, and list of designations for a task environment, to find a good set of differences, expressed in DPL, to be incorporated in GPS for effective problem solving in that environment.

It would not be hard to design such a learning program based on the simple trial-and-error prototype. The program would generate sequences of processes in DPL, and test these for their usefulness as differences. Because DPL has a simple structure, most such sequences would be viable programs, and perhaps a significant proportion of them might even be interpretable, in some sense, as differences. The learning program would incorporate such sequences, tentatively, in the performance program and keep statistics on their application in successful attempts on problems. If a difference were tried and found wanting, the learning program would remove it to make way for a new candidate. Gradually, in the fullness of time and with a non-hostile task environment, the learning program might evolve a satisfactory set of differences.

This much can be done, but we have no reason for <u>a priori</u> confidence — or even hope — that the learning will be accomplished within a reasonable time span. The space of possible differences is very large, and humans guide their trial-and-error searches through it with a variety of heuristics. If the learning program is to operate in real time, it must make use of additional information in selecting and testing candidates

for the set of differences. There is, in fact, a great deal of information available in the task situation, if the learning program can get access to it: information about the structure of the operators; information about the elementary processes of DPL; information about criteria for a good set of differences; information about the transformations that particular operators produce on particular objects in the task environment; and so on. The learning program might even conduct investigations to obtain additional information: for example, it might explore the designation processes in the task environment to discover their mutual relations.

If it is to use such information fruitfully, the learning program must be intelligent — it must be a problem solver. It is doubtful that a simple process, like the one we used to construct the table of connections, exists in this situation. A learning program that resembled a chess-playing or theorem-proving program would have a better chance of succeeding. At any rate, this is our hunch: that an intelligent learning program will differ from a problem solver, not in its structure, but only in the content of its task.

Since GPS is a problem-solving program having pretensions of generality, we can try to use GPS itself as the learning program. If we can restate the learning problem as a problem involving the application of operators to objects in order to remove differences, then, upon presentation of a suitable goal, GPS should be able to work on the new (learning) problem of

creating good sets of differences for the original task environment, and should be able to bring to bear on this learning problem its full repertoire of heuristics.

The idea of using a single "intelligent" program to bootstrap itself appeals to deeply-rooted notions about the reflexiveness that is involved in self-organization. But the strategy has other attractive features. First, it provides a test of the power of GPS and a source of ideas for expanding its repertoire of goals and methods. Second, if the program for learning on a single aspect of performance is as large and complex as the performance program itself, only by using the same program in both roles can we hope to keep the size of the total system within tolerable bounds. This argument becomes even more compelling when we consider the problems of learning on all the other aspects of each performance program.

Our task, then, in the remainder of the paper is to attempt to translate this learning situation into GPS terms, and to evaluate the chances that GPS can handle the learning problem successfully. We will proceed by setting up each of several GPS task environments that seem to be required, and defining the objects, operators, and differences in them. Our own goal, in exploring this path, was to create enough mechanisms to allow us to hand simulate GPS in the process of learning differences. Thus we could provide some assurance that all the essential parts had been identified. We achieved this, but at the cost of a large amount of detail. In the pages that

follow we will give just enough of this detail to convey the general form of the solution and to allow a meaningful sketch of the hand simulation. Since this is not nearly enough information to allow anyone else to verify what we have done, we put the scheme forward in a very tentative spirit.

Basic Task Environment for Learning

From now on we will be applying GPS to several task environments. Since all of them will be formally similar — involving objects, differences, and operators — we need to label them if we are to avoid confusion. We will introduce two of these, the A-environment and the B-environment, at the outset.

The A-environment. We shall call the original task environment (e.g., logic) the A-environment. The A-environment will have A-objects (logic expressions), A-operators (rules), and a list of A-designations (e.g., the test for a connective). The learning problem is to find a good set of A-differences (like the set in the table of connections).

The B-environment. We shall call the environment of the initial learning problem (to learn a good set of A-differences) the B-environment. The B-environment will have B-objects (sets of A-differences), and the learning problem is to find a B-object that makes for good problem solving in the A-environment. Our task is to create B-operators (operators for creating and modifying sets of A-differences), and B-differences (tests for comparing B-objects), and to discover how to state the learning goal as a B-goal. The B-differences must be independent of the

particular task environment, the A-environment, and must use only information derivable from the list of A-designation processes, A-operators, and samples of A-objects.

The B-operators work on sets of A-differences. They add A-differences to a set, delete A-differences from a set, or modify existing members of a set. Since what is available to the learning program are samples of A-objects and A-operators, B-operators are needed that construct A-differences on the basis of their behavior for given samples — i.e., that produce A-differences defined extensionally. Figure 11 gives five such B-operators, enough for the purposes of this paper. Except for Q4, each seeks to obtain an A-difference that gives a specified result, + or ϕ, for a specified pair of objects. (The + is a conventional symbol that means that the A-difference "holds" for the pair of A-objects — that is, gives some non-null output.) These B-operators are applicable to any A-difference in the set, just as in logic or algebra, the commutative law may be applicable to several parts of an expression. The particular element to which a B-operator is to be applied is determined by B-differences, which we shall consider in a moment, or by additional selective heuristics that we shall discuss in more detail a little later. Defining operators by giving the properties of the things they produce does not guarantee that such operators exist, or that, if found, they will accomplish what is wanted of them.

Before we consider the problem of constructing the

THE B-ENVIRONMENT

B-Operators

Q1 Add an A-difference that gives + for pair X and ϕ for pair Y
(A pair may either be a pair of objects, or the condition and product forms of an operator.)

Q2 Modify A-difference T to give + for pair X.

Q3 Modify A-difference T to give ϕ for pair X.

Q4 Delete A-difference T from set S.

Q5 Add an A-difference that gives + for pair X.

B-Differences

D1 The set of A-differences not consistently defined for some pair of objects.

D2 The set of A-operators with no associated difference.

D3 The set of A-object pairs with no associated difference.

D4 The set of non-orthogonal situations (each situation consists of an A-object, a list of A-operators, the product from applying the operators to the given A-object, and the new differences between the input and output that are not associated with any of the operators).

D5 The set of full A-differences (having all A-operators associated with them).

D6 The set of empty A-differences (having no A-operators associated with them).

D7 The set of A-differences with more than one associated A-operator.

D8 The set of A-operators with more than one associated A-difference.

D9 The total number of A-differences.

Table of Connections for B-Operators and B-Differences

	Q1	Q2	Q3	Q4	Q5
D1		x	x	x	
D2	x	x			
D3		x			x
D4		x	x		
D5			x	x	
D6		x		x	
D7			x	x	
D8			x	x	
D9				x	

Figure 11

B-operators, let us look at the B-goals and the B-differences. The highest B-goal for the learning process is a criterion for a good set of A-differences. Ultimately, a good set of differences is one that is effective for problem solving in the A-environment. But to permit intelligent learning, other ways must be found to characterize good sets of differences, so that GPS, in its learning efforts, can evaluate the improvement it is achieving.* We shall provide, in the B-environment, the following criteria of a good set of differences:

1. Only one or a few A-operators should be relevant to each A-difference in the set.

2. Only one or a few differences should be associated with each operator.

3. Each operator should be relevant to at least one difference.

4. Each pair of non-identical A-objects should exhibit at least one A-difference in the set.

5. The set of A-differences should be nearly orthogonal — that is, if only difference D is relevant to a particular operator, then application of this operator to an object should produce only difference D between the input object and the output object.

6. An A-difference should always give the same result when applied to the same A-objects.

The rationale for these criteria is rather simple. The

*We will not consider whether GPS could itself construct the intermediate criteria given only the ultimate performance criteria and experience in several task environments.

differences are the diagnostic tests that GPS uses to determine what operators it should apply in a given situation. The diagnosis will be most efficient if each difference points to the application of one and only one operator, if each operator affects one and only one difference, if the effect of an operator is predictable, and if a difference is always detectable between nonidentical objects. In the limiting case, with a "perfect" set of differences, the performance program would have the trivial task of finding the differences between a given and a desired object, and applying, in sequence the (unique) operators for removing the several differences. Of course, in general, no such perfect set of differences exists or can be found; the task of the learning program is to approximate it as closely as possible. Moreover, there may be more than one satisfactory set of differences. Any such set is a theory of the important features of the task environment and their interrelations.

B-differences, in the light of this discussion of goals, are simply features of sets of A-differences that describe in what respect those sets meet or fail to meet the above criteria. Figure 11 states these differences in measurable form, and gives the table of connections between the B-differences and the B-operators. To weld all the separate "reduce difference" goals in the B-environment into a single effective goal, we establish a priority order of the differences, giving highest priority to completeness properties. (They are so ordered in

Figure 11). GPS will be instructed to produce a set of A-differences, attending first to consistency and completeness requirements. Once these are satisfied, GPS will attempt to improve the set with respect to the lower-priority criteria, always returning to the higher criteria if these are no longer satisfied after the set has been modified.

Let us summarize what we have said up to this point about the learning task. The task is defined in the B-environment, which is an environment suitable for GPS. The B-objects are sets of differences in the A-environment; the B-operators, shown in Figure 11, permit manipulation of these sets; and the B-differences, also shown in Figure 11, correspond to various criteria for "good" sets. The learner's goal is not to attain a fixed, given, B-object, but to construct a series of B-objects in an attempt to reduce the B-differences. We have left open the problem of how the B-operators are to provide differences satisfying the criteria specified. The B-operators assume there is some effective way of programming in DPL to provide suitable A-differences. We now turn to this problem.

Task Environment for A-Differences

If GPS is to construct DPL programs for A-differences, then programming in DPL must be described as a GPS-type task. Again, we need an environment in which this task can be performed.

The C-environment. The natural environment for the task is one where C-objects are DPL programs. Then the C-operators

are ways of putting programs together; the C-differences are things that can be noticed about programs, such as whether one contains a D (difference) or not, and the C-goals are set up by the B-operators: to transform the basic set of programs (the given C-objects) into a program (a new C-object) with certain features.

However, a different environment may be considered. Programs consist of sequences of subprograms — ultimately of sequences of the primitive DPL processes. A program takes an input and transforms it step by step until it finally is made into the output. This sequential character suggests an environment where the objects are the various inputs and outputs, and the operators the elementary DPL processes. Then the final desired program is the sequence that transforms an initial input to a final output.

The situation here is exactly analogous to that in theorem-proving. In symbolic logic, for example, the problem is stated: Prove theorem T, given axioms A, B, ... What is wanted is a proof. But instead of working in the space of proofs — that is, in an environment where proofs are objects — one works in an environment where logic expressions are objects, rules of inference are operators, and the desired proof is the sequence of rules that is applied to get from the axioms (the given objects) to the desired theorem (the final object).*

*Seen in this light, the original LT program [3] was itself a program writer, which generated a program — the sequence of methods leading to the proof — that would produce logic expressions from other logic expressions. It was, of course, an uninteresting programmer, since the product has no particular usefulness as a program, but it provides a direct model for the present self-programming scheme.

We will work entirely in this latter environment, which we will call the D-environment. Our reason for mentioning the C-environment is to explain the ways in which GPS must be extended to work in this D-environment. Not all the relevant information can be obtained by examining the inputs and outputs of programs. An important part of the problem-solving information comes from the properties of programs, viewed as objects (e.g., whether the program contains some task-environment processes or only general processes). Thus GPS must consider not only differences between inputs and outputs, but features of the sequence it is building to bridge the gap — that is, differences that properly belong to the C-environment. The situation is again analogous to theorem-proving where one may impose such constraints as finding the shortest proof, finding an elegant proof, or finding a proof using a given theorem.

The D-environment. Let us define the D-environment more carefully. D-objects are the inputs and outputs of DPL programs. A-objects are included, since these form the initial inputs to the A-differences. All the intermediate products are also included among the D-objects: A-objects with parts replaced by ϕ's (from B[x]), sets and lists of D-objects, the sumbols ϕ and +, and so on. No circularity arises from inclusion of the A-objects, since the only information available in the D-environment about the A-objects is that already available to GPS for learning: the list of environmental processes with their input and output types as shown in Figure 7.

By definition, the D-operators are DPL programs: they transform D-objects into other D-objects. The set of D-operators includes all of the DPL primitive processes, as given in Figures 7, 8, and 9 – some 42 operators in all. It will be noticed that a number of operators contain free variables, whose values are other operators. This is true, for example, of the five operators of Figure 9, which only specify ways of combining other operators.

Beside the D-operators indicated above, we need operators that allow other manipulations of DPL programs than merely adding a new operator to the front end of a sequence of DPL operators. These additional operators are C-operators, properly speaking. However, for the simulation only two such operators were required: one that deleted the last D-operator of a sequence, thus going back "one step;" and one that deleted a D-operator satisfying certain conditions from the middle of the sequence.*

The D-differences are based partly on features of D-objects. GPS can only detect features of D-objects within the limits of the information available for learning. Considered as a D-

*Constructing programs by "working forward" in the D-environment, adding one process at a time, is directly analogous to the way by which the completed DPL program will carry out its information processing. This "analoging" is a very common human technique for programming. But this is not the only consideration that goes into human programming, and we suspect that for programming tasks more complicated than our elementary example, other C-operators will be needed.

object, an A-object is an unanalysable unit, and the D-differences must treat it as such. GPS is able, of course, to examine the sets and lists of objects and parts of objects that are created in the course of a DPL program. Nevertheless, the features of D-objects it detects are rather general:

> Type: GPS can tell whether it is working with an object, a set of objects, a set of sets of connectives, and so on. A glance at the D-operators in Figures 7 and 8 shows that D-operators vary considerably as to the type of D-object taken as input, and the type produced as output.
>
> Size: It is possible to count how big a D-object is, taking as the unit the innermost component. Thus GPS can assign size 3 to a list of three objects, or size 21 to a set of seven sets of three variables each, and so on. It cannot assign a size measure to single A-objects, of course, and must treat them all as equivalent.
>
> Variety: It is also possible to measure the variety of a D-object — to count how many different things there are in the D-object. This is possible because the process, D, provides a test of identity. Variety is a useful notion because certain D-operators, such as B[X], decrease the variety without changing the size of a D-object.
>
> Sign: Finally, GPS can determine whether a D-object is + or ϕ. (Formally this can be done by applying the operator A[+]U to the D-object. This produces the output + unless the object is ϕ, (ϕ,ϕ), ..., in which case it produces ϕ).

The D-differences are also based partly on features of the DPL programs that produce a D-object. Again, these features lead to C-differences, properly speaking. The features we will need are the following:

Set of environmental processes: GPS can note which of the processes that occur in the DPL program also occur on the list of environmental processes given GPS to characterize a particular A-environment (Figure 7).

Set of general processes: GPS can note, similarly, what general processes occur in a DPL program.

Set of special processes: GPS can tell if the program contains some special process, like the constant operator K[X].

Contains a D: GPS can note whether the program contains a D anywhere. This is a very important feature, since this one operator takes a difference between two objects, and thus must be a constituent of every A-difference.

Consistency: A number of DPL processes involve selections from sets — e.g., C, which simply selects a member of the set to which it is applied. Which member is selected, within the constraints laid down by the process, is a matter of "chance." It often happens that the final output of a program is critically dependent on such a chancy event. Thus an A-difference may sometimes give +, sometimes ∅ when applied to the same pair of A-objects. We assume GPS can detect such inconsistencies.

Given these features of D-objects and DPL programs, we can construct D-differences, based on the ability of the various D-operators to change one feature into another. Instead of laying out the table of D-differences, which is rather large and complicated, we will content ourselves with indicating, as we discuss the simulation, those D-differences that played an important role.

It remains to describe the topmost D-goals, which are set up by the B-operators. Consider the B-operator Q5: Add an A-difference that gives + for the pair X. In the D-environment this can be phrased as: Transform X into +. However, there

are some important side conditions. First the DPL program that transforms X to + must be a difference. This can be expressed by requiring the program to contain a D.* Second, the program should not be trivial. After all, the constant function will yield + if applied to an A-object. We can give a partial list of excluded special programs. Taking these points into account, a more complete formulation of the D-goal corresponding to Q5 would be: Transform X into +, in such a way that the transform is a difference and is not trivial. This goal implies a slight generalization from the form of the transform goal given at the beginning of the paper. There we said: Transform object A into object B. Here we allow ourselves to require that additional conditions be satisfied. There is no difficulty in doing this, however, as long as GPS can recognize the existence of unsatisfied conditions and can set up differences and reduce goals based on them.

A glance at the other B-operators shows that similar formulations hold for each one, except Q4, which has no side conditions. Q1 requires a single transform that accomplishes two transformations simultaneously. This is important, since Q1 is the B-operator that brings about a discrimination. Both Q2

*An appropriate operational definition of "difference" should also require that the two inputs to D be dependent on the two input A-objects, and that the output of the program be dependent on the output of D. This dependency can be measured structurally by drawing the oriented graph corresponding to the information flow through the program. Since these conditions never affected the simulation, we indicate them only in passing.

and Q3 require that the transform goal start with a partially completed sequence, although modifications are allowed in this initial segment.

The D-environment is now complete. It differs sufficiently from the initial environments of GPS to require some additional heuristics. The need arises from the large number of D-operators that satisfy various differences. This multiplicity of operators makes further subselection both necessary and profitable. The selection is accomplished in two ways:

1. In selecting an operator, GPS will also consider feasibility: that is, it will match the input type of the operator to the type of the object.

This added test will reduce the attempts to fit infeasible operators. This is reasonable in a situation in which feasible operators are always available.

2. If more than one operator is available after selection by a series of criteria, then the original goal will be consulted and the next most important difference will be generated to provide an additional means of selection.

Thus at each search for a D-operator selection may take place on a number of criteria. Given this opportunity for multiple selection, we can influence the construction of DPL programs by adding further conditions to the goals set up by the B-operators. Besides requiring that the final program be an A-difference, we can also require that it contain some task environment processes, but that it have few of these in common with the other difference programs already in the set. These requirements are distinctly heuristic, for their aim is to

bias the order in which programs are constructed. By giving the heuristic conditions low priority we allow their use occasionally as selective principles when there is lots of choice available, but assure that they will not override more crucial conditions.

Simulation of Learning

We have now described the task environments that will let GPS work on the problem of learning its own sets of differences. This has required a considerable amount of specification: a new programming language, and three GPS environments. And, although we have been fairly specific in describing the language, operators and differences, a number of gaps still exist. Hopefully, however, these are gaps of detail, all the essential mechanisms have appeared.

To shed some light on the question of completeness — which is crucial in an initial exploration — we tried to simulate the program by hand. We took logic as the A-environment, for which GPS was to produce a set of A-differences, corresponding to part of those in Figure 6. The course of this simulation is given in Figure 12. It is very crude. Answers to many questions of detail were created on the spot during the simulation. Many arbitrary selections were made, often without a formal scheme.

The simulation was based on the simple example used earlier to illustrate GPS (Figure 2). The four operators that were chosen, R1, R2, R5, and R6, are the ones involved in that

example; and when a sample problem was needed at step 4 of the hand simulation, that was the one used.

Starting from scratch in Step 1, GPS used operator Q1 to insure that the resulting A-difference would discriminate something. The exploration in the D-environment is shown for this goal, to find an A-difference that is + for R1 and ∅ for R2. What occurred was rather simple: The initial program, consisting simply of process D, was obtained because of the high priority given (by the D-differences) to programs containing process D. Each partial sequence, D, LD, tLD, and so on, produced + with both R1 and R2. Therefore, new processes were added that "decreased size" or "decreased variety" until finally, with the addition of $S[-]$, a program was produced that gave ∅ with R1 and + with R2. This output was just the opposite of what was needed; but the discrepancy was detected by a "reversal" difference and the routine $\overline{A}[+]$ was applied to change the + to ∅ and the ∅ to +. The various branches that were generated but not further explored were rejected either because the output was identical with the input, thus indicating no progress, or because the outputs from both R1 and R2 were identical, so that the process did not discriminate between these two rules.

The question of consistency, which seems a rather technical detail, produced the next phase of the simulation. When T9 was applied to R6 it sometimes gave + and sometimes ∅. When operator Q2 was invoked to remedy this, it produced two

SKETCH OF HAND SIMULATION WITH LOGIC AS THE A-ENVIRONMENT

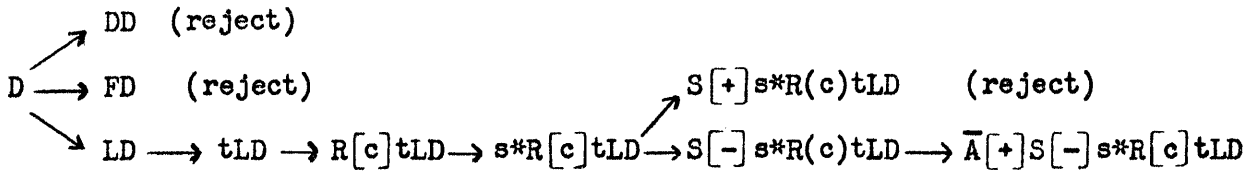
The set of A-operators for which A-differences are to be found is:

R1: AvB —> BvA or A.B —> B.A
R2: A⊃B —> -B⊃ -A
R5: AvB —> -(-A.-B) or A.B —> -(-Av-B)
R6: A⊃B —> -AvB or AvB —> -A⊃B

The initial set of A-differences is the null set:

S0: ϕ

1. The set, S0, is operator incomplete (see D2 in Figure 11) since none of the operators now have associated differences. The first goal given the D-environment is to find an A-difference that is + for R1 and ϕ for R2. The following exploration is conducted in the D-environment:



This gives the next set, S1, of A-differences:

		R1	R2	R5	R6
S1:	T9 = Ā[+]S[-]s*R[c]tLD	+	ϕ	ϕ	?

2. T9 is inconsistent with R6--it gives + or ϕ depending on arbitrary selective processes. The next goal in the D-environment is to modify T9 to produce + for R6. This is accomplished, giving the next set:

		R1	R2	R5	R6
S2:	T11 = S[-]s*tLD	ϕ	+	+	+

3. R1 is no longer covered by S2, so the next goal in the D-environment is to create an A-difference that is + on R1 and ϕ on R5. This leads to the set, S3:

		R1	R2	R5	R6
S3:	T11 = S̲[-]s*tLD	ϕ	+	+	+
	T15 = Ā[+]D(1,r):D	+	ϕ	ϕ	+

4. With S3, all operators have associated A-differences, and all the A-differences are consistent. In order to see if S3 could distinguish non-identical pairs of objects, the simple problem of Figure 2, Transform S.(-P⊃Q) into (QvP).S, was attacked with the above table of connections. T15 was + for the pair of objects, and R1 was applied, just as in Figure 2. However, no difference was found between the left side of L2 and the left side of L4, even though they are not identical. The next problem for the D-environment was to find an A-difference that would produce + for this pair, (-P⊃Q, QvP). This resulted in the next set:

		R1	R2	R5	R6
S4:	T11 = S̲[-]s*tLD	ϕ	+	+	+
	T15 = Ā[+]D(1,r):D	+	ϕ	ϕ	+
	T20 = Dc*	ϕ	ϕ	+	+

S4 distinguishes all the pairs of non-identical objects generated in solving the test problem. The simulation was terminated at this point.

Figure 12

applications of C-operators. The first of these deleted $R[c]$ from the middle of T9, where this process had been identified as the culprit causing the inconsistency. $R[c]$ was identified by tracing through the flow of information. Deletion of $R[c]$ yielded a program (T10) that was ϕ on R6. This discrepancy was detected by the "reversal" difference and the $\overline{A}[+]$ was stripped off the front of T10, giving T11 an A-difference that satisfied the goal.

The change from T9 to T11 left R1 not covered by any A-difference. Q1 was again applied to get an A-difference that would be + on R1. Since a second pair of objects was needed to specify Q1 completely, the input and output forms of R5 were chosen arbitrarily as the second pair. The result of the problem solving in the D-environment was T15. This test was developed in a similar manner to that which produced T9. However, the heuristic of choosing different task environment processes for the two tests resulted in the occurrence of $\underline{r}$ and $\underline{1}$ as components of T15. Again the "reversal" difference accounts for the $\overline{A}[+]$ at the front of T15.

The new set of differences, S3, had all operators covered. It was necessary, next, to use some A-objects in order to test whether S3 could discriminate pairs of non-identical objects. Using the sample problem of Figure 2, a pair of objects, $(-P \supset Q)$ and (QvP), was found that were not identical, but still yielded ϕ when T11 and T15 were applied to them. This caused the next problem-solving attempt in the D-environment, defined

by operator Q5: to discriminate these two objects. T20 was found to do this. It was added to the set to make S4. This final set proved satisfactory for the remainder of the sample problem, and the simulation was terminated. If the simulation had continued, either more problems would have been generated to test for object coverage and orthogonality, or some lesser differences would have been tried in order to improve the discriminability of the table.

Conclusion

We conclude this paper by listing some observations — both reassuring and discomforting — on the path we have followed.

1. The rough simulation presents good evidence, we think, that we have specified the mechanisms that are essential to permitting GPS to work on its own learning. These mechanisms delineate at least one variety of intelligent learning.

2. A feature that stands out clearly in the program is the interaction between the two environments, B and D, one providing the goals for the other. This makes good sense in the light of our general knowledge of computer coding. The distinctions commonly made between "programming" and "coding," and between "problem-oriented languages" and "machine-oriented languages," may reflect the relation between the two environments.

3. An interesting feature of the learning task is that the set of differences is a very ambiguous object. No difference can be completely evaluated in isolation, since the

properties of the set as a whole determine how effective the problem solving is. Similarly, complete factorization with each operator associated uniquely with a particular difference seems unattainable. Hence, the goal of obtaining a satisfactory set of differences, unlike GPS goals considered previously, is not a search for a unique specified object. This difficulty was circumvented by the form of the Reduce goals (Q1 to Q5), which give GPS "direction" in its learning task without specifying a definite final resting place.

4. A considerable amount of mechanism has been added to GPS, but none of this new mechanism seems peculiar to the learning of logic, and much of it, such as the additional selection mechanism, is of the same generality and spirit as the initial version of GPS.

5. In spite of this apparent generality, the justification for much of the mechanism rests on the possibility of using it for a number of different task environments. This possibility is quite untested, for all our work here has been limited to the task of logic. Although GPS has pretensions of being general, only two task environments (logic and elementary algebra with trigonometric functions) have been specified with sufficient precision to exhibit in detail the set of differences. Hence a question that occurs prior to testing this learning program is whether a sufficient population of environments can be constructed in which GPS can operate.

6. A more serious problem is that a general set of

differences may possibly exist that would be effective for all environments. Some of the differences in the logic situation — the set of different kinds of things, and identity of symbols of a given class, like connectives — have a very basic ring. In trigonometry, the one other environment for which we have good information, exactly the same set of differences was used as for logic, with the exception of commutativity and associativity, which were incorporated in the structure of the objects as in normal algebraic notation. If such a "universal" set of differences existed, it might still leave the task of applying differences to the specific environment, but this is a task more like learning the table of connections than like the learning task we have just analyzed. Supporting this possibility is the fact that it is difficult to imagine how differences like commutativity could be built up from more elementary notions. In the simulation this is achieved by the A-difference T15, which compares the left hand side of one object with the right hand side of the other.

7. A final oddity of the present scheme was noted earlier: Different task environments for GPS are completely independent, much more so than the different task environments for a human, which all occur, in the last analysis, in the same real world, to be perceived through the same set of senses. A closer analogy to the human situation would result in GPS if all the task environments were basically analyzable by elementary programs in the general part of DPL. These would

be programs for exploring the structure of objects and making tests of identity on the contents found in various places. Such a scheme not only seems more natural than the present one, but reinforces the vague feeling that there should exist a good set of "universal" differences.

BIBLIOGRAPHY

1. Dinneen, G. P., "Programming Pattern Recognition," Proceedings of the 1955 Western Joint Computer Conference, IRE, March, 1955.

2. Friedberg, R. M., "A Learning Machine," Part I, IBM Journal of Research and Development, January, 1958.

3. Newell, A. and H. A. Simon, "The Logic Theory Machine," Transactions on Information Theory, Vol. IT-2, No.3, September, 1956.

4. Newell, A., J. C. Shaw and H. A. Simon, The Processes of Creative Thinking, The RAND Corporation, Paper P-1320, August, 1958.

5. Newell, A., J. C. Shaw and H. A. Simon, Report on a General Problem-Solving Program, The RAND Corporation, Paper P-1584, January, 1959.

6. Oettinger, A. G., "Simple Learning by a Digital Computer," Proceedings of the Association for Computing Machinery, September, 1952.

7. Selfridge, O. G., "Pattern Recognition and Modern Computers," Proceedings of the 1955 Western Joint Computer Conference, Institute of Radio Engineers, March, 1955.