Reprinted from INFORMATION AND CONTROL, Volume 7, No. 1, March 1964

INFORMATION AND CONTROL 7, 55–77 (1964)

Programs for Machine Learning. Part II*

AIKO M. HORMANN

System Development Corporation, Santa Monica, California

Part I of this paper described the community unit as one of the indirect and implicit means we employ in specifying the behavior of a proposed learning system. It has been pointed out that, for complex problems, the community unit's vision tends to be too narrow and restricted because of its piecemeal manner of attacking problems.

This second part of the paper describes a planning mechanism which attempts to overcome this difficulty by taking a larger view of a given task. After surveying the task in general, the planning mechanism subdivides the task into a hierarchy of subtasks each by itself presumably easier to perform than the original task. This hierarchy of subtasks comprises a rough sketch of a possible course of action which guides the community unit.

To manage classes of problems and to make efficient use of past experience, an induction mechanism is proposed. The induction mechanism will take a still larger view by considering the system's past experience with various problems and by attempting to apply that experience to related problems which have not previously been encountered.

PLANNING: PLACING GUIDEPOSTS ON THE ROAD TO THE GOAL

By planning as an aid to problem solving, we mean analyzing a given problem into a number of smaller subproblems. People confronted with difficult problems typically find planning useful, indeed sometimes indispensable. Planning makes possible more economical search through an immense space of possible combinations, for combinations which will serve as solutions to the given problem. The acceptable combinations are generally scattered throughout the space without apparent pattern and with low frequency in any one sub-space. Minsky (1961)[1] arguing cogently for the power and importance of planning, says that "generally

* Part I appeared in *Inform. and Control* **5**, 347–367 (1962).

[1] Additional discussions and useful suggestions in this area are found in Amarel (1962), Minsky (1957), and in Newell *et al.* (1959, 1960).

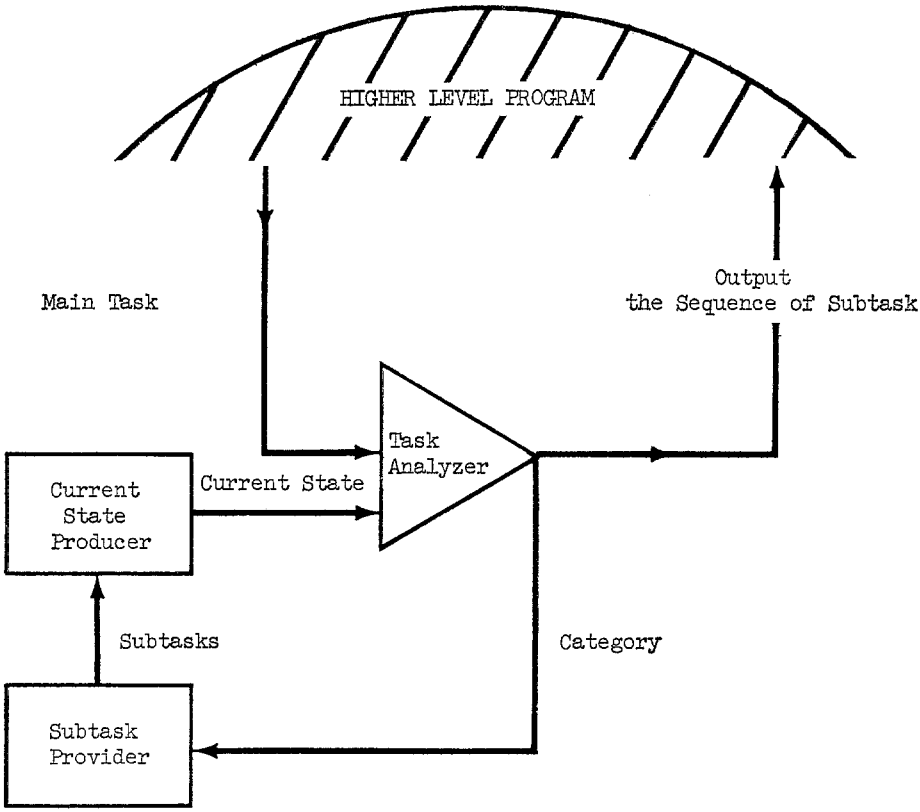

FIG. 1. Planning mechanism

speaking, a successful division [of a complicated problem into a number of subproblems] will reduce the search time not by a mere fraction, but by a *fractional exponent.* In a graph with 10 branches descending from each node, a 20-step search might involve 10^{20} trials, which is out of the question, while the insertion of just four lemmas or sequential subgoals might reduce the search to only 5×10^4 trials, which is within reason for machine exploration.... Note that even if one encountered, say, 10^6 failures of such procedures before success, one would still have gained a factor of perhaps 10^{10} in over-all trial reduction!"

A planning mechanism which views, at an abstract level, a given task as a whole and then suggests possible divisions of it into a number of subtasks, each of which can be attacked by a smaller search (or be further divided), is proposed for the system.

Proposed Mechanism

Our planning mechanism (Fig. 1) is similar in structure to the community unit. In fact, the planning machanism uses, in addition to its own records, the same record in memory which the community unit uses. We again assume that requested tasks are in descriptive form. (See Part I, pp. 349–350.)

We propose to use a set of characterizing expressions (attributes and their values) such that a particular subset of this set serves to define a task category. Categorization of a task for the planning mechanism may be fine or coarse depending on the amount of detail, i.e., on the size of the subset of characterizing expressions. Associated with each category, fine or coarse, are names of methods or operations which will probably help perform a task belonging to a category. For each associated method or operation there is listed a probable utility value and a description of the method in descriptive request form. That description shows the input (current state) and output (desired state) of the method or operation. The coarser the task category, the more abstract and general the corresponding descriptions of methods or operations will be; descriptions of input and output will contain parameters and will indicate only what is likely to happen when a method or operation is applied to a task belonging to the category.

Following task analysis and categorization by the task analyzer, the subtask provider proposes to the current-state producer, another part of the planning mechanism (Fig. 1), a set of subtasks in the form of methods or operations with input and output expressed in descriptive form. The input of a proposed method or operation must somehow be similar to the current state of the given task. Criteria for similarity are relaxed or tightened depending on the coarseness of task categorization being used. The current-state producer uses the output descriptions of the proposed subtasks to determine current states of new tasks and lists possible values for each parameter if there are any. The task analyzer records the output of the current-state producer as branches of the state graph as shown below and chooses one of them as defining the next task to be analyzed. The choice is made on the basis of externally provided criteria. For instance, the choice might be made on the basis of some rough measure of "how far" each proposed current state is from the desired state of the original task. (Cf. The Newell, Shaw, and Simon General Problem Solver.) Given the new task, defined by the chosen new current state and the originally given desired state, the same sequence of steps is repeated.

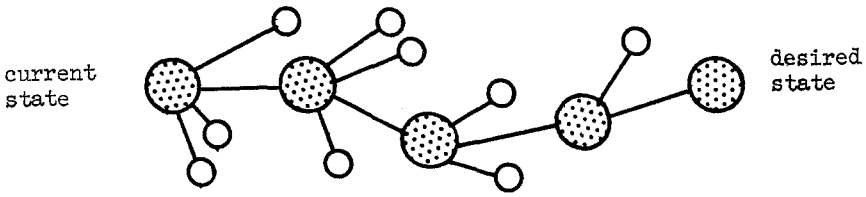

FIG. 2. State graph at a planning stage

Shaded nodes of the graph (Fig. 2) indicate grouping of "similar" states using coarse task categorization during the early planning stages. Under the control of a higher-level program which specifies the level of detail at each planning stage, a rough plan is developed first with only a few guideposts inserted; then subplans are developed connecting the guideposts (nodes on the graph, taken two at a time consecutively upon re-entering the planning mechanism) and using finer categories; then subsubplans are developed using still finer categories, etc.

Objects manipulated by the planning mechanism are state descriptions representing stepping stones; actual operations which make these state changes possible are not the direct concern of the planning mechanism. It should be noted that although the process of performing the given task eventually has to be discovered in detail, in executable form, fine details and exact matches need not be sought until a reasonably good plan is obtained.

Subtasks, defined by pairs of states, are matched roughly with abstracted input-output descriptions of available methods or operations belonging to the category. Matching current states of proposed subtasks with the desired state of a previously chosen subtask is done only roughly in the early planning stages. However, as the planning progresses and subtasks are divided into smaller subtasks, similarity criteria are tightened and the match of states must be made more and more exact. As the descriptions of states or of subtasks become more concrete and more detailed, the system may ascertain that methods or operations specified for a particular subtask do not exist in the system's memory. Depending on the estimate of the difficulty of the subtask, which depends on the number of unmatched parameter values and their positions, a higher-level program decides whether or not to (1) activate the community unit to work on this particular subtask by modifying existing programs whose abstracted description previously matched the state descriptions of the subtask, (2) look for a possibility of further sub-

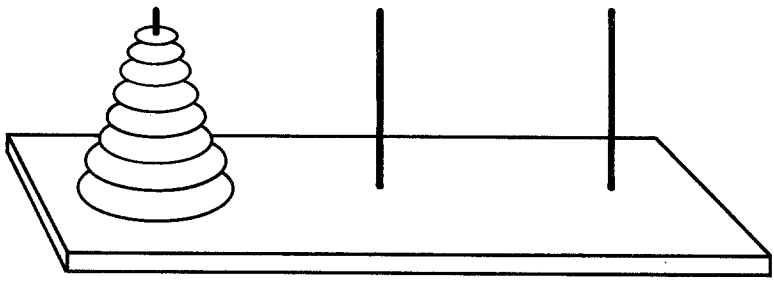

FIG. 3. The Tower of Hanoi puzzle

division, (3) modify the current plan by considering other possible subtasks to replace this subtask, possibly resulting in changes in subsequent subtasks, or (4) back up one level of planning stages by relaxing the similarity criteria and coarsening the categories in order to start a new direction of exploration and to proceed with planning again. Note that in case (1), the abstracted description of methods or operations, which have been matched previously with the subtask, already narrowed the search for programs to which the required program must be "similar." Furthermore, places where matching fails after finer details are supplied, automatically indicate where the modification effort should be focused.

AN ILLUSTRATION OF PLANNING MECHANISM OPERATION

TOWER OF HANOI PUZZLE

A puzzle called the "Tower of Hanoi"[2] is used here to illustrate some of the features discussed in connection with the planning mechanism. We shall first introduce the puzzle independently of the mechanism.

The problem posed in the Tower of Hanoi, illustrated in Fig. 3, is to transfer the tower of disks from one peg to either of two empty pegs in the fewest possible moves, moving one disk at a time and never placing a disk on top of a smaller one. It has been proved that the fewest possible moves for n disks is $2^n - 1$.

This puzzle was chosen as the first testing vehicle for our system for several reasons. Its solution is relatively simple but not trivial. In an experiment using human subjects, the time required for solution ranged from minutes to days. For some it was unsolvable. Another reason for choosing the Tower of Hanoi puzzle is that the solution is known to the

[2] GARDNER, MARTIN, (1959), "Mathematical Puzzles and Diversions," pp. 57-59. Simon & Schuster, New York.

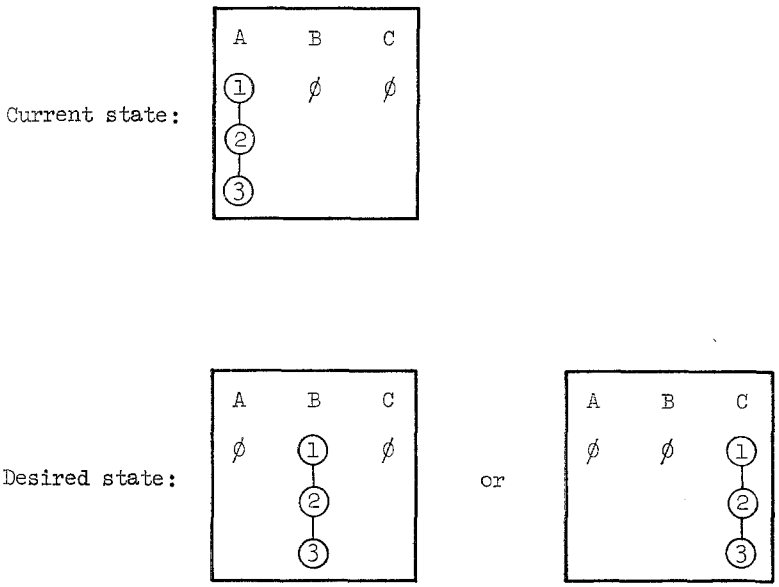

FIG. 4. 3-Disk case in the descriptive request form

experimenter, so that evaluation of performance is easier. In addition, the puzzle can be varied by altering the number of disks and pegs (currently we always use three pegs) thus allowing a training sequence from the simple to the more difficult within the same class of tasks. The puzzle also has the important property that the methods for simple cases, with suitable abstraction, do provide some help in solving harder cases in a fairly non-trivial way.

The puzzle was given to the system in the descriptive request form. Figure 4 illustrates the task for the three-disk case. Rules of the game were presented in the form of a program which generates legal moves when the current state is given.

Columns A, B, and C in the request form diagrams above represent

the three pegs in the puzzle and the circled ①, ②, and ③ are numbered disks from the smallest to the largest. The symbol ϕ indicates the column is empty. Both states are stored as list structures, i.e., the current state is a list whose elements are A, B, and C; A itself is a list made up of the elements ①, ②, and ③; and B and C are empty lists prior to the first move.

If there are more alternatives than one for the desired state as shown above, this is internally indicated by a code and by having two or more elements in a list named "desired states." Each element is then expanded as a list. For the Tower of Hanoi experiment, we varied the problem either by giving two alternatives for the desired state or by making the desired state unique.

Figure 5 depicts the list structure (move tree) for some of the legal moves generated by the program. There are always three legal moves at each node of the move tree but only two are indicated because the third one just reverses the move which leads into the node. The top line of each box shows the symbolic representation of the particular move; for example, $1B$ means "Move disk ① to column B." The bottom of the box shows the current state of the puzzle *after* the move is made. Heavy lines indicate a minimal path; for the three-disk puzzle as stated, there are two minimal paths, one ending with three disks in column B, the other in column C (not shown in Fig. 5).

How many nodes (moves and current-state configurations) exist for an exhaustive search? It can be shown that the total number of nodes in the complete tree for n disks is $\sum_{i=1}^{2^n-1} 2^i = 2^{2^n} - 2$. This gives us 254 for the three-disk case, 65,534 for the four-disk case, 4,294,967,294 for five disks, and 18,446,744,073,709,551,614 for six.

Planning for the Tower of Hanoi Puzzle

Suppose the system is currently given the four-disk case after having accomplished the three-disk case, and channels this information to the planning mechanism. Figure 6 is a schematic representation of a task given in descriptive form. The task analyzer uses the system's abstraction routine to find that the given task has one more element, ④, than the previously accomplished task, everything else being identical. It then uses the system's replace routine to represent three disks, ①, ②, and ③, by one element.

The subtask provider, using a legal move generator, produces both of two possible legal moves (M_1 and M_2 in Fig. 7). Note that three disks are moved together as if they were one disk even though it is illegal to

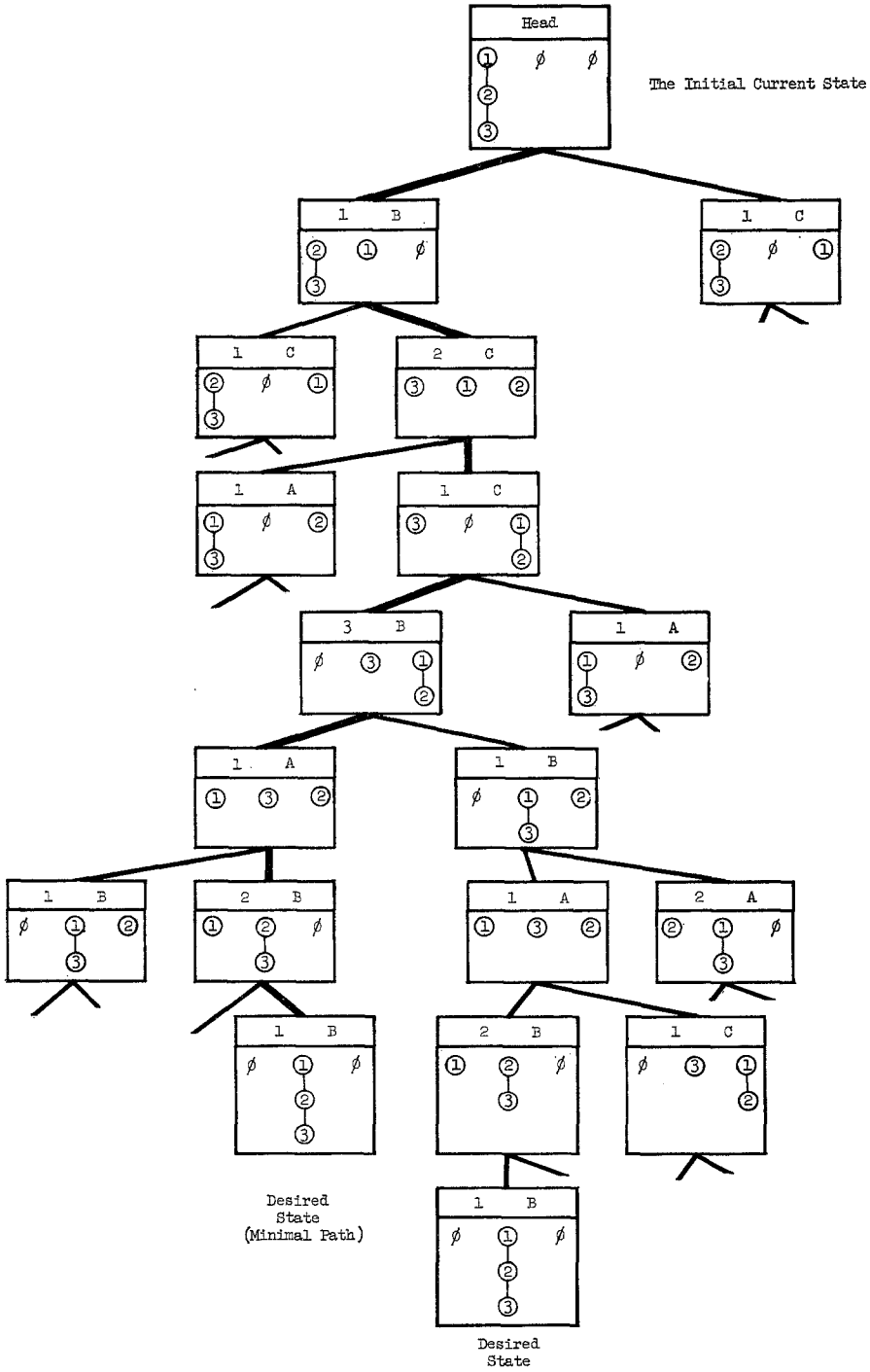

FIG. 5. Part of move tree for the 3-disk puzzle

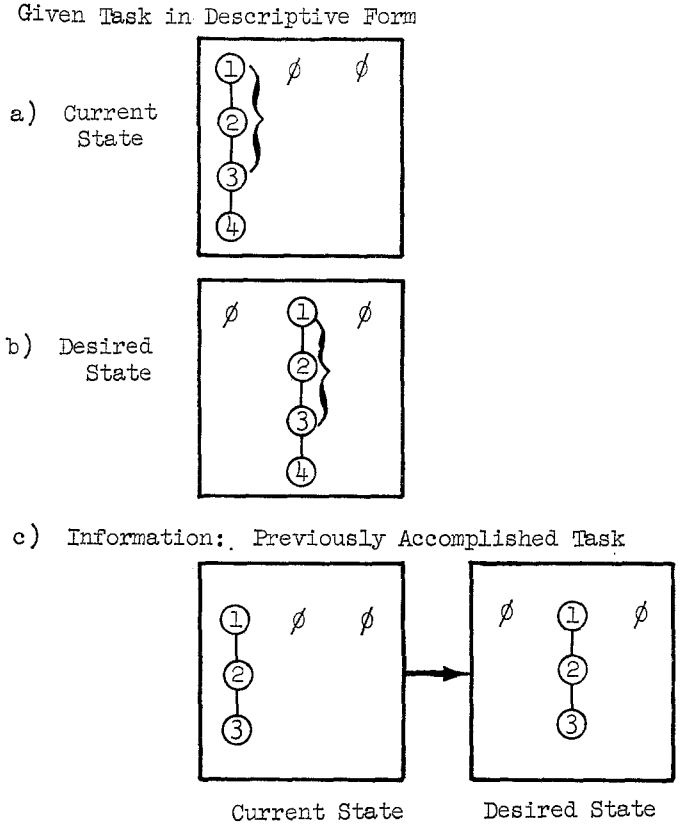

FIG. 6. Tower of Hanoi puzzle given to the planning mechanism

move more than one disk at a time. This is done in the planning stage only because the system has a record of previous accomplishment in the three-disk case and because moving three disks from one peg to another can ultimately be expressed in terms of legal moves. This is analogous, in the community unit, to a subroutine being treated as a single operation even though it may be composed of a number of elementary operations.

Each M_i is stored by the mechanism as a subtask in descriptive request form, the top part of an arrow in Fig. 7 pointing to its current state and the bottom part pointing to its desired state. From these subtasks, the current-state producer finds and outputs the desired states to the task analyzer as current states of new subtasks. The task analyzer chooses

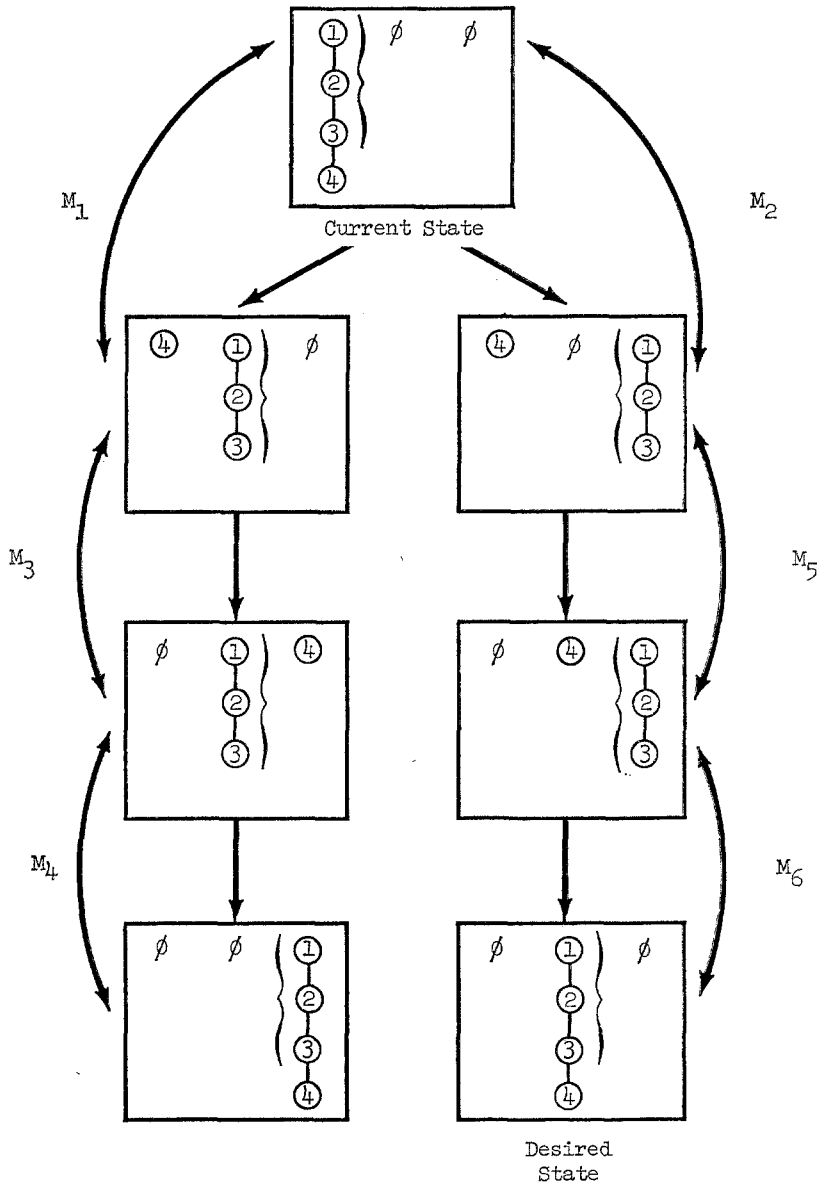

FIG. 7. Subtasks of Tower of Hanoi puzzle

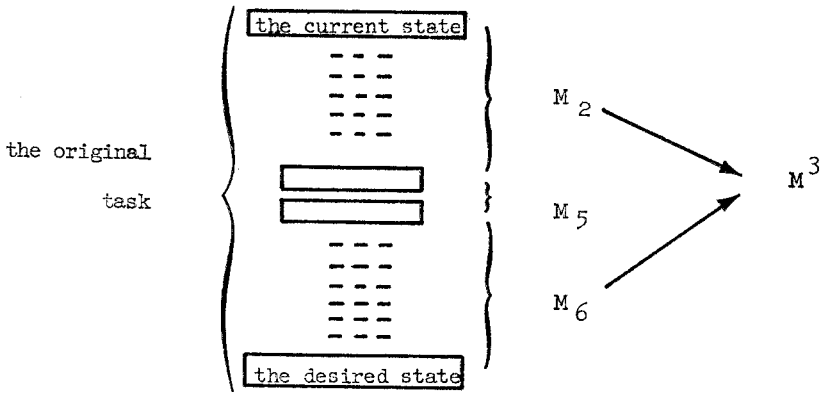

FIG. 8. A plan for the 4-disk case

M_1 instead of M_2 because of the information on the previous task. But this choice, leading through M_3 and M_4, does not work out because it does not lead to the right end state. The planning mechanism returns to M_2 and thus finds its way to M_5 and M_6.

The final sequence of subtasks is presented schematically in Fig. 8.

The original task is represented by the space from the top to the bottom rectangles (Fig. 8), and the gaps, indicated by M_2, M_5, and M_6, are subtasks created by the planning mechanism. The broken horizontal lines indicate steps yet to be filled in. The current and the desired states for both M_2 and M_6 do not match those of the task previously accomplished; the operation which worked before is for three disks starting at column A as the current state and ending at column B as the desired state. These "before" and "after" states of the known operation are then abstracted so that column names can be unspecified. At this level of abstraction, even though only one variation is known, all six variations[3] of the three-disk puzzle are treated the same and are solved, and are represented by an abstracted form we will call M^3. Successful accomplishment of M_2 and M_6 now requires instantiating on the abstract form, i.e., supplying values to parameters. The instantiation is not a matter of trial and error but is directed by the requirement of exactly matching current and desired states of the sequentially ordered

[3] For each n-disk case, there are six variations; for the current state, there are three possible column positions and for each of these, the desired state may take one of the two remaining column positions.

subtasks. For our example, the actions necessary for accomplishment of M_2 and M_6 are significantly fewer (in number of operations and execution time) than if they were performed without using the system's past experience in one of the six variations.

In the example, the planning mechanism has to examine all possible subtasks[4] before it finds the right sequence. This is an exhaustive search. However, planning of this kind is relatively cheap—it takes six examinations at this planning stage to find the path. But if we were to consider individual moves instead of the larger steps we know how to make on the basis of past experience, an exhaustive search for the correct path at this stage would take 65,534 examinations.

A CASE OF MECHANICAL INDUCTION

Induction may be defined as the formulation of general rules about observed cases of a phenomenon and the application of these rules to the making of predictions. There are a number of useful articles discussing inductive inference from the standpoint of artificial intelligence, e.g., Amarel (1962), Kochen (1960) Solomonoff (1957, 1960), and Watanabe (1960).

The inductive procedure observed in humans may be described in general terms: when a human wants to formulate general rules about a class of phenomena, he first makes a guess to form a hypothesis; next he deduces certain consequences of his hypothesis and tests them against new and old evidence; and then he increases his confidence in the hypothesis, modifies the hypothesis, or forms a new hypothesis and repeats the procedure.

For our experiment, we give to the proposed system, as a training sequence, simple inductive tasks. A set of general rules to be formulated by the system is unique and is known to the experimenter, so that he can provide the system with information about the degree of its success and can suggest lines of investigation which may result in the modification of a previously formed hypothesis or in the forming of a new hypothesis, by the system.

The mechanism proposed has a structure similar to that of the com-

[4] There are two legal moves generated by the legal move generator at each node, but the other moves possible instead of M_3 and M_5 involve moving of the group ①-②-③ just moved (see Fig. 7). We assume here, for simplicity, that the system has learned or has been told that moving of the same item twice in succession is wasteful because a single move can obtain the same result.

munity unit. Here we shall present it in the context of solving one particular task, but it is to be hoped that at least some useful generalization beyond this particular task is possible. We say this despite claims that the main difficulty with artificial-intelligence research is that it cannot generalize beyond the very specific tasks for which programs are written and systems designed. (See (Kelly and Selfridge, 1962).)

The inductive task we examine is that in connection with the Tower of Hanoi puzzle. The system is given a sequence of tasks of increasing difficulty and is asked to discover how to solve the puzzle for n disks when methods of solving the puzzle for 3, 4, $\cdots$, $n - 1$ disks are known.

We have set up two experiments with this puzzle. The second experiment is a more realistically designed extension of the first experiment; the system is given only the basic "knowledge" of the puzzle itself and three "hints," two of which are helpful only a part of the time, and is asked to find a successful sequence of moves for the three-disk case. A great deal of trial-and-error action results at first, but the manner of growing and examining move-trees becomes less and less aimless as the system gains experience with the three-disk case, the four-disk case, the five, etc., until finally, with this induction mechanism, the general pattern of successful moves for any n-disk case is discovered.

Since the first experiment, although rather unrealistically restricted, makes direct use of the induction mechanism, it will be described in detail. A "mechanical trainer," which knows everything about the puzzle, is stored outside the learning system in the computer memory; its purpose is to eliminate actual human intervention as much as possible in order to use computer time efficiently.

The trainer simply "gives" one of the successful sequences of moves for the three-disk case together with the corresponding task representation in the descriptive request form. Let us employ a shorthand notation, $H(n, A, B)$, for the case in which n disks are transferred from column A to column B, and $H(n, A, B/C)$ to indicate the disjunctive case having two alternatives B and C for its desired state.

The first information given to the system is that the task is $H(3, A, B/C)$ and the successful sequence of moves which accomplishes $H(3, A, B)$ is

$$\begin{array}{ccccccc} 1 & 2 & 1 & 3 & 1 & 2 & 1 \\ B & C & C & B & A & B & B \end{array},$$

where the top line shows particular disks moved and the bottom line

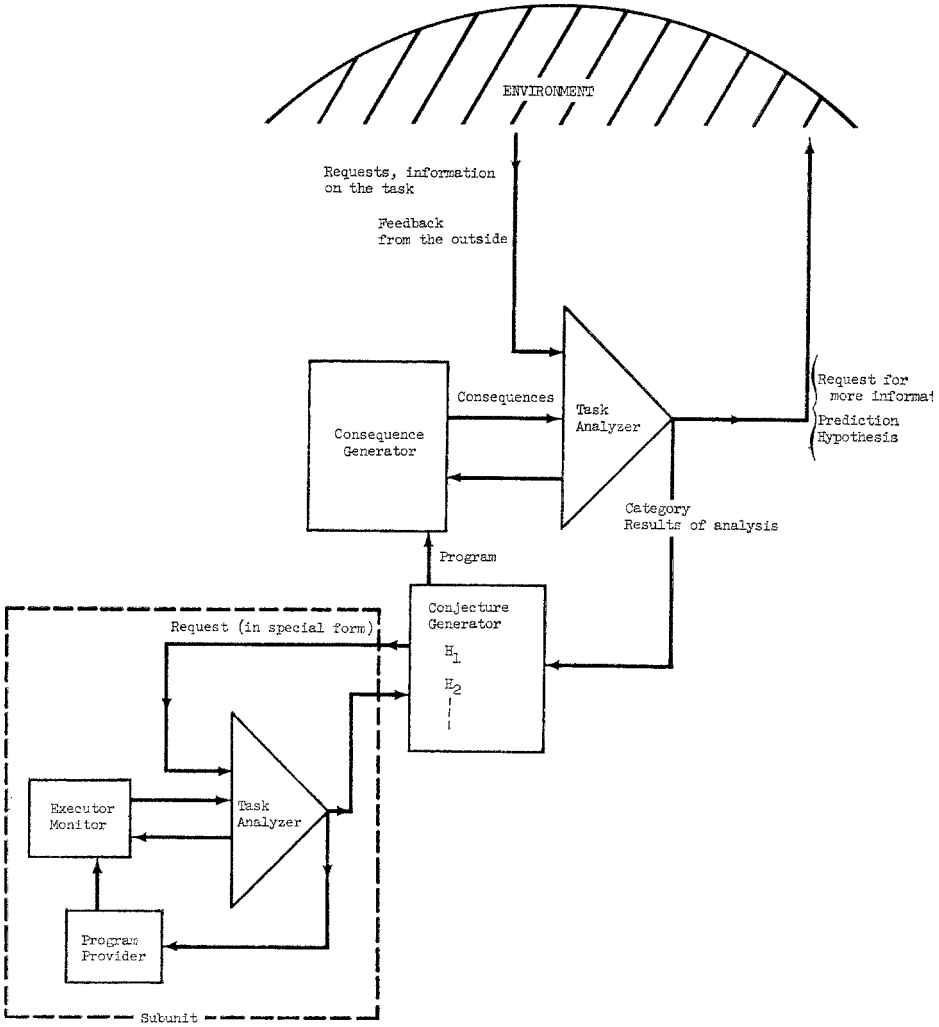

FIG. 9. Induction mechanism

shows the names of columns to which these disks are moved. For instance, $\frac{1}{B}$ means that disk ① was moved to column B.

The four-disk case $H(4, A, B/C)$ is now given, and the system is asked to find the successful sequence for the new case. The induction

mechanism then goes to work. Figure 9 shows the structure of the mechanism with its members: the task analyzer, the conjecture generator, and the consequence generator.

Observation and Analysis

The first phase of any inductive process is performed by the task analyzer of the induction mechanism. For the Tower of Hanoi puzzle, the task analyzer begins by comparing the descriptive request form of the three-disk and four-disk cases (see Fig. 10) by means of the abstraction routine. The conclusion is that both cases are identical except for the additional disk, disk ④, in both the current and desired states of the four-disk case. For the next step the elements in the successful sequence of moves for the three-disk case and the elements appearing in the description of the case are compared by the abstraction routine. The conclusion is that the kind and number of distinct elements are the same, i.e., that the third abstracts of both contain A, B, C, ①, ②, and ③ and nothing else. The conjecture is made that the sequence of moves for the four-disk case must contain the additional element ④ if it is to match the elements appearing in the description of the four-disk case. Furthermore, in examining the sequence of moves, it is discovered that

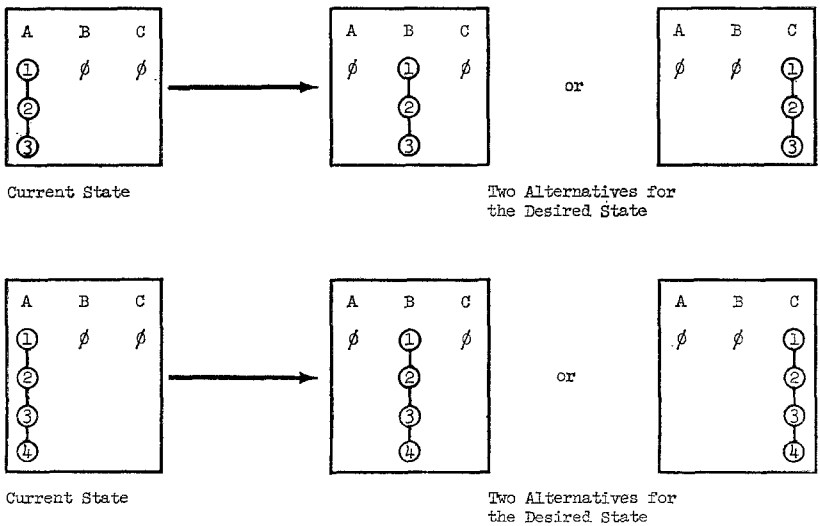

Fig. 10. 3 and 4-disk puzzles

some elements appear more than once. The task analyzer now outputs the results of its analyses and a category, "cyclic," to the conjecture generator.

Conjecture Generation

Using the information from the task analyzer, the conjecture generator, with the aid of its own subunit, produces programs which represent conjectures (see Fig. 9). The requests, which the conjecture generator constructs and gives to its program-generating subunit, constitute a special case of the generalized request form different from the special case given as the descriptive request form. Each request is represented by a list which contains three pieces of information:

(1) A sequence of symbols in a list form.

(2) A symbol indicating that the desired output is a program to *regenerate* the sequence given in (1).

(3) Symbols representing restrictions and characterization on the task which the task analyzer of the induction mechanism is able to supply.

In our example, the conjecture generator receives the category, "cyclic," together with the sequence of successful moves for the three-disk case, i.e.,

$$\begin{array}{cccccccc} 1 & 2 & 1 & 3 & 1 & 2 & 1 \\ B & C & C & B & A & B & B \end{array}.$$

The conjecture generator separates the top from the bottom line and makes two requests of its subunit. In each of the requests, the information on the task stipulates that the sequences produced by the generated programs are to have a cyclic pattern.

Performance of the Subunit

We now examine what the subunit does in the case of the top line. Since a cyclic pattern is requested, the task analyzer of the subunit looks for the first recurrent position of the first item on the list "1 2 1 3 1 2 1" and finds it to be the third item. It now takes the first two items "1 2" as defining a cycle phase and asks the program provider to construct a program which will generate "1 2 1 2 1 2 ···"

The program provider now constructs a sequence of instructions which, when executed, would generate the symbols ① and ② repeatedly, and in alternation, any number of times until stopped. This sequence of instructions is then given to the executor-monitor.

Let us examine the interaction between the executor-monitor and the task analyzer when these instructions are monitored. The proposed program, under monitored execution, brings the symbol ① and the symbol ② alternately to the task analyzer for examination. The task analyzer compares each symbol it receives with the corresponding symbol in the given sequence, keeping track of the order and the correspondence. The first three elements, ①, ②, and ①, presented by the program agree with the given sequence, but the fourth one, ②, does not.

Upon detecting this discrepancy, the task analyzer of the subunit looks for the second recurrence of the first item. This turns out to be the fifth item. It then uses the first four items "1 2 1 3" as defining a cycle phase. This time the program provider constructs a program which, when executed, would generate symbols ①, ②, ①, ③ repeatedly.

Interaction between the executor-monitor and the task analyzer this time shows that results agree with the given sequence "1 2 1 3 1 2 1." The task analyzer now outputs the program to its higher-level program, the conjecture generator (see Fig. 9) with a "success" signal. As far as the subunit is concerned, the task is successfully accomplished, although the produced program is proved to be inadequate by the higher-level programs at a later stage.

Notice that in this method of generating a cycle-producing program, the subunit will always find a program which fits the given sequence; it is sure to succeed when it takes the entire given sequence as defining a cycle.

When the same procedure is used for the bottom line of the original information, "*B C C*" is the first cycle phase tried, "*B C C B A*" is the second, and the final accepted one is "*B C C B A B.*" The conjecture generator now combines these two programs so that they will produce together a sequence of pairs of the desired form and outputs the result to the consequence generator.

The Consequence Generator and Its Interaction with the Task Analyzer

The consequence generator, together with the task analyzer, step by step examines programs supplied by the conjecture generator. The examination consists of monitored execution. Each item proposed as a member of the solution sequence is in turn proposed to the environment by the task analyzer as a prediction of the next move needed to solve the four-disk case.

The mechanical trainer, serving as the environment of the mechanism first checks the legality of a suggested move by means of the legal move generator. If illegal, the information is fed back to the task analyzer. If legal, the trainer compares it with its stored "knowledge" of the puzzle, and feeds back information whether the move is right or wrong.

In our example the first seven, the ninth and the eleventh suggested moves turn out to be right but the eighth, tenth and twelfth moves are wrong. A comparison of the suggested and correct moves is:

suggested moves:	1	2	1	3	1	2	1	[3]	1	2	1	3	1	2	1
	B	*C*	*C*	*B*	*A*	*B*	*B*	*C*	*C*	[*B*]	*A*	[*B*]	*B*	*C*	*C*
correct moves:	1	2	1	3	1	2	1	4	1	2	1	3	1	2	1
	B	*C*	*C*	*B*	*A*	*B*	*B*	*C*	*C*	*A*	*A*	*C*	*B*	*C*	*C*

Squared items indicate where the task analyzer is informed of illegal or wrong moves.

After the complete sequence of correct moves becomes known to the task analyzer, the unmatched elements in the suggested sequence and the correct-move sequence are then determined and given to the conjecture generator. The conjecture generator modifies previously constructed programs by parameterization, i.e., it replaces unmatched places with parameters. The resulting programs, when executed, would produce a sequence like this:

1	2	1	P_1	1	2	1	P_1	1	2	—	—	—	—	
B	*C*	*C*	P_2	*A*	P_3	*B*	*C*	*C*	P_2	—	—	—	—	

Underlined parts represent cycle phases. P_1, P_2, and P_3 are names of sublists. P_1 contains ③ and ④, P_2 contains A and B, and P_3 contains B and C. The fact that ④ is used for the four-disk puzzle is consistent with the conjecture made earlier that successful moves for the four-disk case must contain the element ④. Up to this point, however, this conjecture has not been implemented. Our system learns! Next time it immediately makes use of the corresponding conjecture. When the five-disk case is presented, the task analyzer tentatively includes ⑤ as one of the possible values of P_1. Comparison of suggested and correct moves for the five-disk case is:

suggested moves:

1	2	1	P_1	1	2	1	P_1	1	2	1	P_1	1	2	1	P_1	1	2	1	P_1	1	2	1	P_1	1	2	1	P_1	1	2	1	P_1	1	2	1
B	*C*	*C*	P_2	*A*	P_3	*B*	*C*	*C*	P_2	*A*	P_3	*B*	*C*	*C*	P_2	*A*	P_3	*B*	[*C*]	*C*	P_2	*A*	P_3	*B*	*C*	*C*	P_2	*A*	P_3	*B*	*C*	*C*	P_2	*A*	P_3	*B*

correct moves:

1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 5 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
B C C B A B B C C A A C B C C B A B B A C A A B B C C B A B B

When the task analyzer specifies the possible values for the parameters, in every case only one of the possible moves is legal so the correct move is automatically determined without trial and error for each of the parameter positions. This is, of course, a singular feature of the Tower of Hanoi puzzle. If such a convenient feature were not present, a secondary pattern within each list of parameter values would have to be detected, using the same mechanism but at a different level of analysis. Among the moves suggested for the five-disk case, there is only one move which is wrong, a move at a nonparameter position, position 20. The task analyzer gives this information to the conjecture generator which modifies the existing programs so that they will use an additional parameter. The resulting sequence of moves looks like this:

$$\underline{1\ \ 2\ \ 1\ \ P_1}\ 1\ \ 2\ \ 1\ \ P_1\ 1\ \ —\ \ —\ \ —\ \ —$$
$$\underline{B\ \ P_4\ \ C\ \ P_2}\ A\ \ \underline{P_3}\ B\ \ P_4\ \ C\ \ —\ \ —\ \ —\ \ —$$

where P_4 contains A and C and underlined elements indicate cycle phases.

When the new programs are used to suggest moves for the six-disk case, all turn out to be correct. In fact the parameterized program which has now been constructed will solve any n-disk case for three pegs, as long as the current state has n disks in column A and the desired state is disjunctive, i.e., $H(n, A, B/C)$ in our shorthand notation. Of course, the system itself will never know the fact unless told by the trainer. However, as the system gets more and more experience with the puzzle, and the conjecture (the program) is used successfully more and more times, utility values of the conjecture increase so that the task analyzer will tend toward directing a straightforward use of the program.

However, when the system is given the four-disk case, $H(4, A, B)$ with B indicating a desired state differing from that of the previously accomplished task, $H(4, A, C)$, the situation changes. The task analyzer must undergo more analyses and formulate a new conjecture, although it can make use of the previously formulated, already successful conjecture. This particular case is not described here since it involves very little participation of the induction mechanism.

SUMMARY ON THE INDUCTION MECHANISM

A few important features of the induction mechanism deserve emphasis:

1. Parameterization in the abstraction process is one way to separate more relevant from less relevant information *without ignoring the latter*. In our example, the second pattern was first suggested as $B\ C\ C\ B\ A\ B$, next it was parameterized to $B\ C\ C\ P_2\ A\ P_3$, and finally to $B\ P_4\ C\ P_2\ A\ P_3$. At each stage, constants indicate items which are unaffected by the change of task. Finally an unchanging pattern is revealed.

2. Two-level usage of the feedback structure permits the initial ad hoc manner of generating conjectures to become less arbitrary each time the mechanism is given more information. Note that the program-generating subunit is requested to *regenerate* a given sequence under conditions imposed by the conjecture generator; the subunit simply obeys. The given sequence and conditions may change each time the subunit is used, but such changes are decided by the conjecture generator, not by the subunit. Decisions made by the conjecture generator are influenced by analyses made by the task analyzer which, in turn, are influenced by higher-level programs.

3. Conjectures are represented by executable programs. The conjecture program is executed and tested *directly* while it is being formed by the subunit and also while it is being used to generate consequences. A program which embodies a generating principle provides a compact and direct means of representing the inductive process of extrapolating beyond recorded instances.

4. What about problems whose complexity is beyond the direct reach of such a mechanism? Suppose, at the higher level, the system can observe the function of the induction mechanism and the way in which puzzles have been presented from the simpler to the more complex, ultimately resulting in a general workable strategy for n-disk puzzles. It is extremely important that the system be able to imitate the over-all process in the future.

Students in natural sciences often learn, by imitation, clever heuristics for finding suitable simplification. They observe scientists making deliberate oversimplifications of a situation by considering only a few variables and by restricting the behavior of these variables to simple known functions. Scientists usually study simple cases first and then vary them to more complex cases, study the effects of changes, make conjectures, and repeat the process. If our system had learned these

processes, and if the eight-disk case of the Tower of Hanoi puzzle were given at the outset, it might have tried out the two-disk or three-disk puzzle of its own accord. Note that at this stage, solving the puzzle even by the exhaustive method is feasible. For the three-disk puzzle, the exhaustive method would require 254 examinations of the current state configuration whereas such a method for the eight-disk case would be out of the question.

In order for the system to be able to learn from a carefully selected training sequence and use the experience toward creating its own trial sequence of simplified tasks, the system must be able to construct and modify its "cognitive map" with temporal sense. In addition, effective utilization of the cognitive map is necessary; this may be realized by a special higher-level program, "master monitor," which *ruminates* periodically and takes a bigger view of the tasks given in the past rather than focusing on one task at any given moment.

CONCLUSION

A system of programs with three mechanisms has been proposed. To discover capabilities and limitations of such a system, a study is being made to see how it works in specific, relatively simple situations. We shall try several experiments to see in what ways the system falls short of the intended "learning system." There can be no doubt that before we can achieve such a system there is a great deal of learning *we* must do.

We begin with a simple system and give it simple tasks. It is our hope that we shall discover principles applicable to a complex system which can work at different levels of abstraction as well as in different problem situations. We are aware, however, that methods which work on simple cases may not necessarily work on more complex ones.

We are interested in discovering *how* higher-order composite capabilities might evolve from a given set of a priori capabilities. Our interest, however, is *not* in discovering what can be made to evolve from a *minimum endowment.* If a powerful learning mechanism becomes available, we shall probably want to preprogram the system to the limit of our capabilities before we turn it loose.

External Feedback: Communication between the System and Its Trainer

Most of our discussion of feedback has been in terms of internal communication among units and subsystems. We assumed only a limited

amount and a very restricted form of feedback from outside the system. Ultimately, however, we wish to give the system lessons, exercises, and hints in much the same way as we do for human learners. McCarthy (1959) points out, "In order for a program to be capable of learning something it must first be capable of being told it." The main difficulty in computer communication is that the transformation of descriptive information into *instructions* is not a simple trick, as it seems in human communication. It is one thing to "tell" the machine what to do by inputting programs and letting it execute them, but it is another to "describe" what is to be done and expect the machine to do it. Executable programs must somehow be produced by the machine before it can perform the task.

EPILOGUE

This epilogue describes the current status of the implementation of the system by means of IPL-V programs.

Both experiments with the Tower of Hanoi puzzle mentioned in the section on the induction mechanism have been successfully completed. Members of the community unit have been programmed and the monitored mode of some operations tested, but none of the sophistications for generalization have been incorporated. Our experience with the Tower of Hanoi puzzle and with the simple version of the community unit indicates that more appropriate internal representation and manipulation schemes are necessary for what amount to construction, modification, and utilization of a "cognitive map."[5] (We have so far used IPL-V for both external and internal languages.)

Before we can start concerted effort toward full-scale implementation of the system, much more research is necessary on techniques related to characterization of tasks and methods and to a distance concept which, in turn, links together other concepts such as similarity judgment, partial success, and difficulty estimates. We are currently working on each mechanism somewhat independently with research emphasis on the techniques mentioned above. However, for more complex problem situations, all mechanisms must be coordinated and work together. A rudimentary attempt at such coordination is described by Hormann, Shaffer and Van Wormer (1963).

RECEIVED: July 13, 1962

[5] See p. 354 in Part I of this paper.

REFERENCES

AMAREL, S. (1962), On the automatic formation of a computer program which represents a theory. *In* M. C. Yovits, G. T. Jacobi, G. D Goldstein, eds., "Self Organizing Systems," pp. 107–175. Spartan, Washington, D. C.

HORMANN, A. M. (1962), Programs for machine learning, Part I. *Inform. and Control* **5**, 347–367.

HORMANN, A. M., SHAFFER, S. S., VAN WORMER, T. A. (1963), Gaku: An Artificial Student of Problem Solving. SDC Document TM–1524, September 16. System Development Corporation, Santa Monica, California.

KELLY, J. L., JR. AND SELFRIDGE, O. G. (1962), Sophistication in computers: a disagreement. *IRE Trans. Inform. Theory* **IT-8**, 78–80.

KOCHEN, M. (1960), Experimental study of "hypothesis formation" by computer. IBM Research Center Report RC-294, May 25. International Business Machines Corporation, Yorktown Heights, New York.

MCCARTHY, J. (1959), Programs with common sense. "Mechanisation of Thought Processes," Vol. 1, pp. 75–84. National Physical Laboratory Symposium No. 10. Her Majesty's Stationery Office, London.

MINSKY, M. (1957), Learning systems and artificial intelligence. "Applications of Logic to Advanced Digital Computer Programming," University of Michigan, College of Engineering, Summer Session.

MINSKY, M. (1961), Steps toward artificial intelligence. *Proc. IRE* **49**, 8–30.

NEWELL, A., SHAW, J. C., AND SIMON, H. A. (1959), A general problem-solving program for a computer. *Computers and Automation* **8**, 10–17.

NEWELL, A., SHAW, J. C., AND SIMON, H. A. (1960), A variety of intelligent learning in a general problem solver. In M. C. YOVITS AND S. CAMERON, eds., "Self Organizing Systems," pp. 153–189. Pergamon Press, London.

SOLOMONOFF, R. J. (1957), An inductive inference machine. *IRE Natl. Conv. Record* **5** (2), 56–62.

SOLOMONOFF, R. J. (1960), A preliminary report on a general theory of inductive inference. Zator ZTB-138; Zator Company, Cambridge, Mass., November.

WATANABE, S. (1960), Information-theoretical aspects of inductive and deductive inference. *IBM J. Res. Develop.* **4**, 208–231.