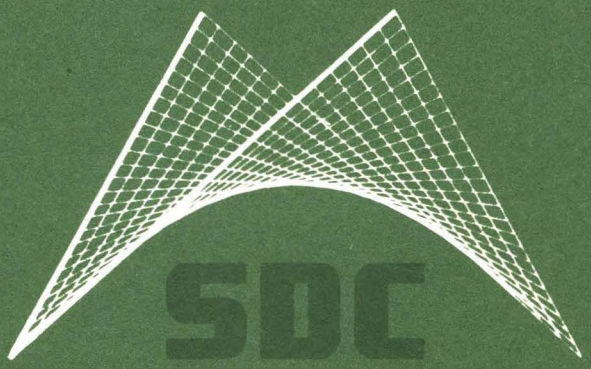


TM-669/000/01

Programs for Machine Learning

29 May 1962

TM-669/000/01

TECHNICAL MEMORANDUM

(TM Series)

This document was produced in connection with a research project sponsored by SDC's independent research program.

Programs for Machine Learning

Aiko M. Hormann

SYSTEM DEVELOPMENT CORPORATION
2500 COLORADO AVE.
SANTA MONICA
CALIFORNIA

May 29, 1962

(TM-669/000/00 is a draft dated April 4, 1962)

TABLE OF CONTENTS

(Page 4 Blank)

LIST OF ILLUSTRATIONS

SUMMARY

This paper reports on a proposed schema and gives some detailed specifications for constructing a learning system by means of programming a computer. We have tried to separate learning processes and problem-solving techniques from specific problem content in order to achieve generality, i.e., in order to achieve a system capable of performing in a wide variety of learning and problem-solving situations.

Programs in the system are given by the programmer either directly or indirectly. Indirectly given programs are those which are constructed inside the system (by a set of programs constituting a program-providing mechanism) from an existing supply of basic instructions and component programs, some of which have been directly given and some of which may have been previously constructed by the system itself. The primary purpose of indirect programming is to assist higher-level programs in performing tasks for which detailed preplanning by an external programmer is either impossible or impractical.

Most of the indirect programming is performed by a mechanism called the community unit. It is presented in a schematized framework as a team of routines connected by first and second-order feedback loops. Analogies are drawn and an illustration of community unit operation is given. Some heuristics are suggested for enabling the community unit to search for a usable sequence of operations more efficiently than if it were to search simply by exhaustive or random trial and error. These heuristics are of a step-by-step nature.

For complex problems, however, such step-by-step heuristics alone will fail unless there is also a mechanism for analyzing problem structure and placing guideposts on the road to the goal. A planning mechanism capable of doing this is proposed. Under the control of a higher-level program which specifies the level of detail required in a plan being developed, this planning mechanism is to break up problems into a hierarchy of subproblems each by itself presumably easier to solve than the original problem.

To manage classes of problems and to make efficient use of past experience, an induction mechanism is proposed. An illustration is given of the induction mechanism solving a specific sequence of tasks.

Parts of the system are currently being programmed and tested in IPL-V on the Philco 2000 computer. Illustrations given in connection with the community unit and the induction mechanism show the results of hand-simulation based on the programming specificiations.

(Page 8 Blank)

ACKNOWLEDGEMENTS

In addition to the material specifically referenced in this paper, clearly I have been influenced by the writings of many other people. Outstanding among such writings are those of Bruner, Polya, and Ashby. My indebtedness to many uncited works of Newell, Shaw, and Simon is also apparent.

I am grateful for several enlightening discussions with Marvin Minsky and Ray Solomonoff. It was Professor Minsky who pointed out the similarity between TOTE units of Miller, Galanter, and Pribram and parts of the mechanisms of the system I propose and discuss. I wish to express my gratitude to Frank Marzocco, Head of the Artificial Intelligence Research Staff at SDC, and to my colleague, Larry Travis, for many useful suggestions and much valuable advice in addition to their editorial assistance.

INTRODUCTION

Our aim is to increase the "intellectual" capacities of machines by directly programming on a computer and, ultimately, to construct what might reasonably be called an intelligent learning system. Consideration of human problem-solving and learning activities permits analogies to be drawn and suggests the use of certain heuristic processes for the machine. The resulting system of programs, however, is not meant to be a model of human thought processes. We hope to find heuristic programs which do not deliberately imitate human characteristics.

We are motivated by the belief that the capacity of a machine might be expanded by means of a learning mechanism to handle increasingly complex and varied tasks. The solution of even well-defined problems for which the mathematical or physical rules are known can be extremely difficult to program. A practical approach might be "preprogramming" to the limits of human ability, then letting the system learn the rest of the techniques required for problem solution. In addition, some ill-defined problems, such as many socioeconomic problems, might be handled effectively and economically by a good learning system.

FEATURES OF THE MECHANISMS: HELPFUL ANALOGIES

The proposed system contains, as its essential components, several mechanisms each having the same general structure. One mechanism, which we call the community unit, is basic to the others. We shall talk about it first.

The function of the community unit is either to provide its customers

(higher-level programs) with a program capable of performing a requested task or to perform a customer-stipulated task by executing a program. If the mechanism does not have a ready-made program in stock to fill a particular request, it will have to construct a program and "debug" it before outputting or executing it. The process of constructing a tentative program, testing, modifying, testing again may have to be repeated many times.

Machine shop analogy

The structure and function of the community unit can best be introduced by means of an analogy. Let us consider a machine shop with general-purpose assembly machinery and a good supply of basic parts, partially assembled parts, and products which can be used either by themselves or as parts. The corresponding community unit supply consists of basic instructions, sequences of instructions or open subroutines, and closed subroutines. The supply in either case is classified into categories which correspond to the job categories into which incoming requests are classified. Machine shop customers may want either a machine, e.g., a grinding machine, or the service of a machine, e.g., grinding a cam shaft.

The chief engineer receives a request from a customer, studies it, identifies the characteristic features of the request, and decides the category. If the customer's request can be satisfied with an "off-the-shelf" item, the chief engineer can fill the request immediately. If the service of an available machine is wanted, the request goes to the design engineer, who sets up the machine; the test-and-service engineer then supervises the machine's performance.

However, we are more interested in the case where the request cannot be filled with machines existing in the shop. The chief engineer passes the request and his categorization of it along to the design engineer. The design engineer designs and constructs a pilot machine from materials available to him. He may modify an "off-the-shelf" item which is finished or partially assembled. He may combine such items, modified or unmodified. He may construct his machine from basic parts only. It is possible that the request cannot be filled because of an inadequate supply of parts or because of insufficient technical know-how, but if the design engineer is able to construct a pilot machine, it is channeled to the test-and-service engineer.

The test-and-service engineer activates the machine step-by-step and at the end of each step reports to the chief engineer (see Figure 1). The chief engineer considers the performance of the pilot machine in the light of his analysis of the customer's request and does one of the following three things. He may inform the test-and-service engineer that the function of the machine is in agreement with the customer's requirements so far and tells him to proceed with further testing. If there is nothing more to test, the chief engineer outputs the machine to the customer who, in turn, either accepts or feeds back information about necessary changes. However the chief engineer may detect some undesirable feature of the machine, in which case he tells the design engineer to modify the machine or construct a new one. Finally the chief engineer may find it necessary to ask the customer for additional information. After the customer examines the

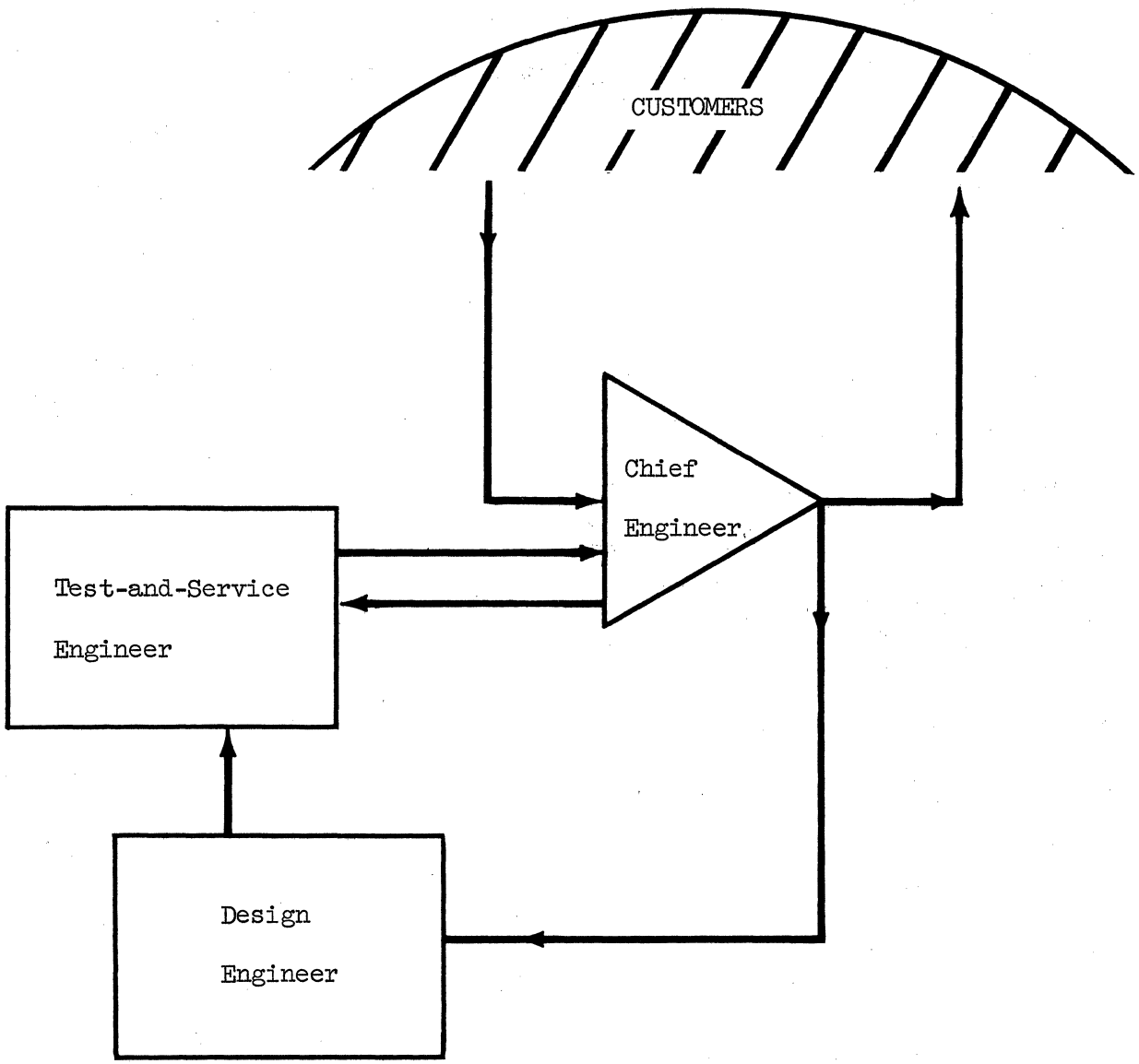


Figure 1.

Machine Shop Example

performance of the pilot machine as reported by the chief engineer, he may get a clearer idea of what he really needs. The customer may be able then to provide more details about what he wants his machine to do; he may even be able to suggest specific modifications of the proposed machine.

The entire process will be repeated until the customer is satisfied or places a stop order. A hierarchical structure results if some customers are themselves engineers in machine shops constructing machines for their own customers. The original machine shop of our illustration can also "subcontract" with another machine shop, requesting parts for the machine under construction. Subcontracting is to be realized in the proposed system by recursive use of the community unit.

It has been pointed out that the customer may request either a machine or the service of a machine. In both cases, we suppose that requests are expressed in a functional sense, rather than in a material sense; i.e., the customer says either, "I want a machine which does so and so," or "I want so and so done." For example, the customer says, "I want this nail to be flush with this board." He does not stipulate that the hammering be done with a light claw hammer or with a sledge hammer or with a rock. The amount of experience and the level of sophistication of the shop determines the means by which the customer's request is met. The essential nature of the shop's operation, however, is the same in every case: first, to convert the functional description of the task into a machine or a program or a system of some kind, i.e., into something that can be described in material terms; and then, in contrast to the first stage, to proceed from the product or a

material description of it, to a functional description which can be directly compared with the original request. Good techniques in both conversions are extremely important to the shop, and engineers must learn to find clever conversion techniques. In addition, engineers must also learn to arrange a good assortment of supplies using past experience in fulfilling a variety of requests. For example, if some frequently requested machines have the same structure and use similar parts, it will be more efficient and economical to keep in stock a "ready-made framework" for such construction and only modify minor parts when a particular machine is requested.

Request forms

The choice of a form for expressing request depends on the subject's past experience. In the machine shop analogy, the customer may know by name a particular product which will meet his needs. If the chief engineer knows the product by the same name and it is an "off-the-shelf" item, the request can be made simply by using the name. However, if the customer does not know a name of an item which will meet his needs or if there is no such item on the shelf, he must make his request in a different form.

The three alternative forms proposed for requests made by or of a community unit can be clarified by another analogy, a hammering task, as shown in Figure 2. We might say to someone, "Hammer this nail." If the person has previous hammering experience or if he has previously watched someone hammering and associated that action with the command, "Hammer," he can carry out the task even though he may hammer somewhat clumsily. On the other hand, he may have no idea what "Hammer" means. In that case, we may

Examples / Request Forms	Human Task to 'hammer'	Community Unit Task to 'exchange'
Appellative	Say 'hammer this nail'.	Name command 'exchange' followed by AB, as a list. (Unless there is the required routine, named 'exchange,' there will be no response.)
Descriptive	Show 1) The nail upright in place on the board. 2) Picture of the nail flush with the board. 3) Up-and-down motion of the arm.	Describe 1) Current State: AB 2) Desired State: BA 3) Information on the task: a code to indicate the desired state is to be stored on top of the Current State.
Instructive	Do action of hammering.	Input (from outside the system) Desired program for the task.

Figure 2.

Forms of expressing requests depending on the subject's past experience.

place the nail upright on the board, show a picture of a nail which is flush with the board as the desired state, and then demonstrate with an up-and-down motion of the hammer. The third alternative is to instruct him in the desired action by hammering the nail ourselves or by taking his hand and making the motion of hammering with him.

Now consider a simple exchange as a specific example of a task to be performed by a community unit. Two symbols, A and B, are currently stored in location L_1 and L_2, respectively. A is to be moved to L_2 and B is to be moved to L_1. The customer, i.e., the controlling program, may request that the community unit perform this task by naming the command "Exchange" followed by its operand A and B. Perhaps the task can be carried out directly, but if an exchange routine has not been generated previously or prestored, there will be no name "Exchange" among the names of available routines. In other words, the request will not be understood. Then the controlling program may input the request in a more descriptive form. This is to be done by giving a list. The first item on the list represents the current state AB, the second item represents the desired state BA, and the third item gives information about the task, i.e., that the state represented by the second item is to replace the state represented by the first item. Notice that each item itself may be a list or a list-structure (7). The information in the third item may be a set of restrictions or conditions and may also include additional information which the controlling program is able to supply about the task. This information may be in the form of programs or in the form of data. If the task cannot be performed as presented in

either of the preceding forms, a third alternative is for the controlling program to output the request in the descriptive form to a human teacher, asking the teacher to input the necessary program. The system then takes the program, assigns a name to it, and stores these with the descriptive form of the request. Thus the correspondence between the appellative and descriptive request forms is established for future use.

THE COMMUNITY UNIT AND ITS MEMBERS

We now describe the community unit (Figure 3) in programming terms. Analogies used in the previous section are referred to when useful. The hammering example is used again to describe a kind of feedback loop called a TOTE (Test-Operate-Test-Exit) unit by Miller, Galanter, and Pribram (4). The function of a TOTE unit is illustrated schematically by a man using sensory feedback as he moves his arm in the hammering task (Figure 4). Similarity and the correspondence between the TOTE and the community unit will be discussed later.

The task analyzer

The task analyzer corresponds to the chief engineer in the machine shop analogy. The task analyzer receives incoming requests either in appellative or descriptive form. We shall consider a request in the latter form, with the unit having no previous experience relevant to the assumed task. Also we shall assume, for simplicity, that the task analyzer has a built-in ability to find characteristic features of the

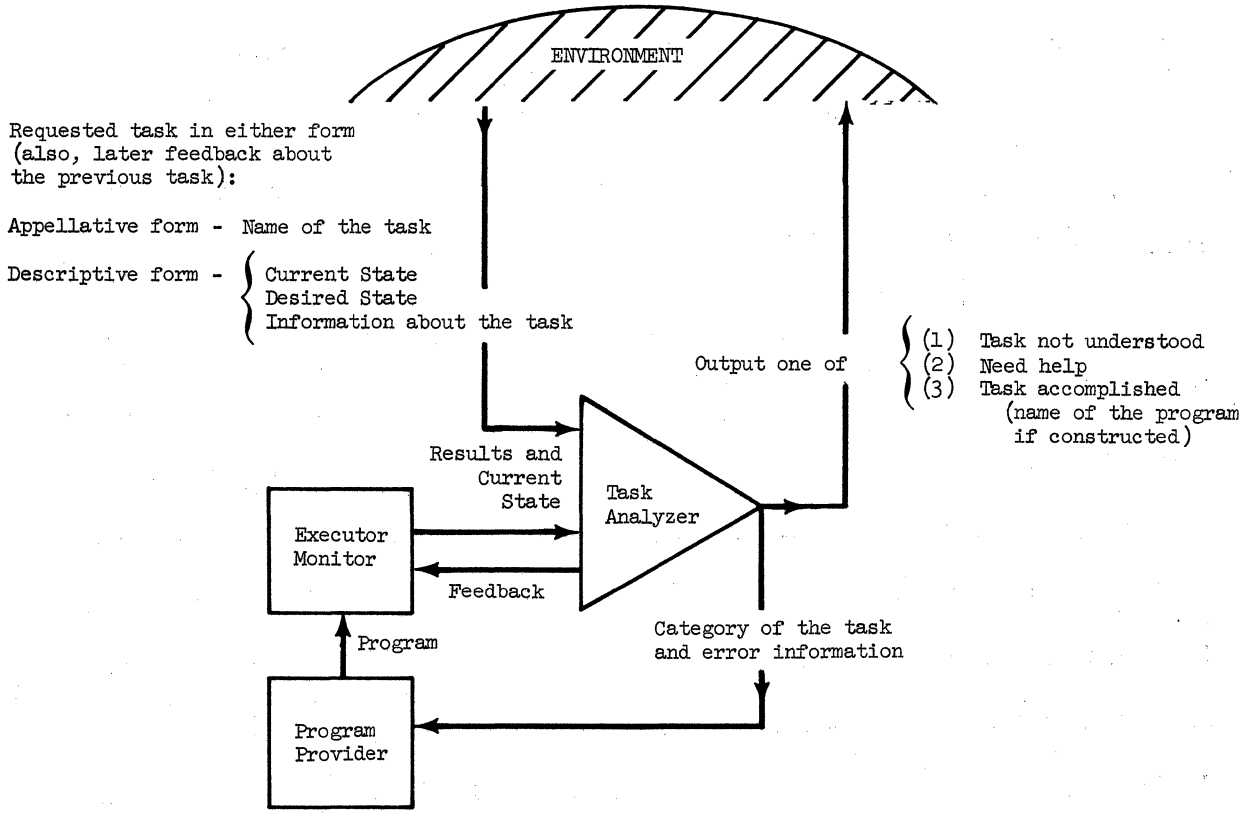


Figure 3. Community Unit

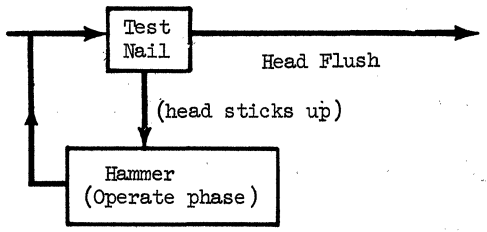


Figure 4. Hammering as a TOTE Unit

task and to determine the proper problem category.[1]

The program provider

The program provider corresponds to the design engineer. The program provider has a collection of available instructions and programs which is divided into two parts: one is a permanent set, a standard repertory, which contains basic instructions, sequences of instructions or open subroutines, and closed subroutines, each represented by a single name; the other is a temporary set consisting of records of previously constructed programs. The manner in which members of the temporary set are abstracted and generalized to become members of the repertory represents an important kind of learning which will be discussed later.

Using the information received from the task analyzer, the program provider constructs a tentative program by modifying a previously stored program, by constructing a program from basic instructions, or by assembling a new program from previously constructed programs, modified or unmodified.

In modifying a previously stored program, a similarity test may reveal that a previously solved task matches the present one closely and the solution developed then may, with modification, work in the present case. In the hammering analogy, a similar task may have been solved with up-and-down arm movements. An attempt may be made to adapt the sequence then developed

[1] The ability to learn to classify problem situations into effective categories is one of the most important capabilities any "intelligent learning machine" must have if it is to use its past experience effectively. Useful suggestions on how such learning might be realized have been made by Minsky (4, 5 and 6) Newell, Shaw and Simon (8 and 9) and Solomonoff (10).

to the task of hammering. So in programming, an old routine can often be modified to suit a new purpose, perhaps simply by changing addresses, loop parameters, branching criteria, etc.

If the simple modification technique is not applicable, it may be necessary to construct a program from basic instructions. In the hammering analogy, "move the arm upward" and "move the arm downward" may have to be further broken down into sequences of contractions and relaxations of particular muscle groups.

Several considerations must be taken into account when a new program is assembled from previously constructed programs as building blocks. If the task can be divided into subtasks, all of which can be recognized as identical to previously solved tasks, then organizing them in a proper sequence is all that is required--though doing this may be no simple task. In the hammering analogy, the motion of the arm can be built up as a sequence of upward and downward motions. It is likely, however, that some of the subtasks will require modifications of previously constructed programs, and for some others, it may be necessary to construct programs from basic instructions. These subtasks, except the ones which have been recognized as identical to previously solved tasks, will be represented by descriptive request forms which will then be input to the community unit one level lower. This corresponds to subcontracting in the machine shop analogy. What happens then is treated below in the section on the executor-monitor.

If the unit is mature, i.e., if it already has had much experience, many requests will be satisfiable by modification and subdivision, and the rest may be simple enough to permit construction from the basic instructions.

In order to avoid having complex requests which cannot be handled by any of the preceding methods, the training sequence has to be selected carefully. If there is a big change in the current request from the previous ones, either in complexity or in the degree to which it or its subproblems are similar to previously satisfied requests, attempts at solution will involve a great deal of trial and error, probably ending in failure.

Whatever the means by which an item is entered in the repertory, associated with it is a separate utility value for each of the problem categories. Thus if there are n categories, each item has n values since the utility value of an item is expected to be different for different categories. In addition, attached to each item in the repertory is a description of the results of each action of that item. Similarly, when a human programmer decides to use a particular instruction or a subroutine in his program, it is usually because he has a clear picture of the before and after states and not necessarily only because the chosen item has a higher utility value than others in a particular category.

Items in the initially given repertory have their descriptions pre-stored, but the repertory changes as the community unit learns; some members of the repertory are combined to become one item, and some members of the temporary set are abstracted and generalized to be added to the repertory. Each time such a change takes place and each time the task analyzer records a change in current state as the result of its interaction with the executor-monitor discussed below, the description of the item

involved must be reviewed and updated.

The act of providing functional descriptions and improving them in the light of experience is an important kind of learning which might be described as constructing and modifying a "cognitive map."[1] It is the utilization of such a cognitive map which enables one to internalize overt action, e.g., considering possible chess moves and, on the basis of information in the cognitive map, internally determining what their consequences would be were they actually to be made.

The term cognitive map is used in a relative sense; the system as a whole, or a part of the system, can develop its cognitive map as it experiences a variety of tasks. The term "environment" will also be used in a relative sense. When only a part of a system is considered, its environment includes the rest of the system.

The cognitive map of the program provider contains functional descriptions of items in the repertory; improvements of its cognitive map mean improvement of its ability to select proper instructions and routines to construct a required program.

There are, however, considerable difficulties in describing the function of every item in the repertory. In order to make the utilization of the cognitive map effective, there must be an efficient system for internal coding with reasonably uniform format. Our current attention is restricted

---

[1] "Cognitive map" is a term of Tolman (12); a similar notion is called "Image" by Miller et al (4).

to the type of description which can be put in the descriptive request form. This is a strong restriction, but its uniform format offers two major advantages. First, when a new program has been constructed to fill a request expressed in descriptive form, the same form can be stored (along with its given name) to serve as the description of the program. Second, when a number of similar programs are abstracted by parameterization, corresponding parameterization of the descriptive form will immediately serve as the description of all of them.

A generalized version of the descriptive request form is introduced later but corresponding generalizations of the community unit function and its cognitive map construction have not yet been determined.

Executor-monitor

Upon receiving the tentative program constructed by the program provider, the executor-monitor begins executing the given program in one or a combination of the following two modes. In normal, high-speed execution of a sequence of instruction, the executor-monitor transfers control to the address of the first instruction of the sequence. All instructions in the sequence will be executed in high-speed[1] and control will not be returned to the executor-monitor until the end of the sequence is reached. This mode of execution is identical to that of the conventional computer.

---

[1] Since the computer plus IPL-V is considered as another computer, the mode of operation of IPL-V is considered as normal high-speed execution even though its speed is not the speed of machine-code execution.

The second mode is monitored execution. Instead of transferring control to the program location the executor-monitor interprets the execution of each instruction. This mode of execution not only permits the executor-monitor to retain complete control during execution but also to use information on the action and the results of each instruction. Thus it can detect a danger before any destructive action takes place. For example, a transfer instruction may go to a data location or some reserved program location or an instruction to store data may refer to a location tagged as containing data required later in the program. While these operations may be correct for this application, they also may not be. In the event a danger is detected, the executor-monitor will record the place of interruption in the program, and transfer control to the task analyzer, outputting a danger signal and the address of the particular operation at which the danger was detected.

The mode of execution depends on the following rules, chosen to ensure that programs are monitored until they have been "debugged." (1) Each of the basic instructions occurring alone in the repertory is always monitored. (2) A sequence of instructions represented by a single name in the repertory, if not modified by a process of the program provider, will be executed in high speed. If it has been modified, every instruction in the sequence will be monitored. (3) Subroutines in the repertory, if not modified by a process of the program provider, will be executed in high-speed, and if modified will be monitored. It should be noted that if there are standard input preparations prior to entering the subroutine, instructions affecting them will be attached to the subroutine in the repertory, but

their execution will always be monitored even though the subroutine may not be. (4) As long as routines and subroutines are in the temporary set, they are always monitored.

Subtasks which are in descriptive form cannot be executed; therefore, these are input to the task analyzer as new requests and the entire community unit becomes involved again. Such recursive use of the community unit is made possible by a push-down list (7) used by the task analyzer; the second entrance to the task analyzer, before exit is made from the first task, does not destroy the information needed for exit from the original task.

Figure 5 illustrates a mixture of the two execution modes. Horizontal lines indicate monitored instructions, and vertical lines indicate high-speed execution. In this example, subroutines A and C are taken to be in the repertory. Subroutines B can be either in the temporary set or in the repertory, but are modified for the given task. A sequence of instructions in the repertory will always be copied in the constructed program since such a sequence will have no entrance and exit provisions of the type used with closed subroutines. Provision must be made, however, for returning to the executor-monitor at the end of the sequence.

Interaction between the executor-monitor and the task analyzer

The executor-monitor and the task analyzer are the only two parts of the community unit with direct two-way communication (see Figure 3). As the executor-monitor executes instructions, the picture of the current state changes. Since the executor-monitor's function is essentially execution, however, and its immediate attention is given only to the current

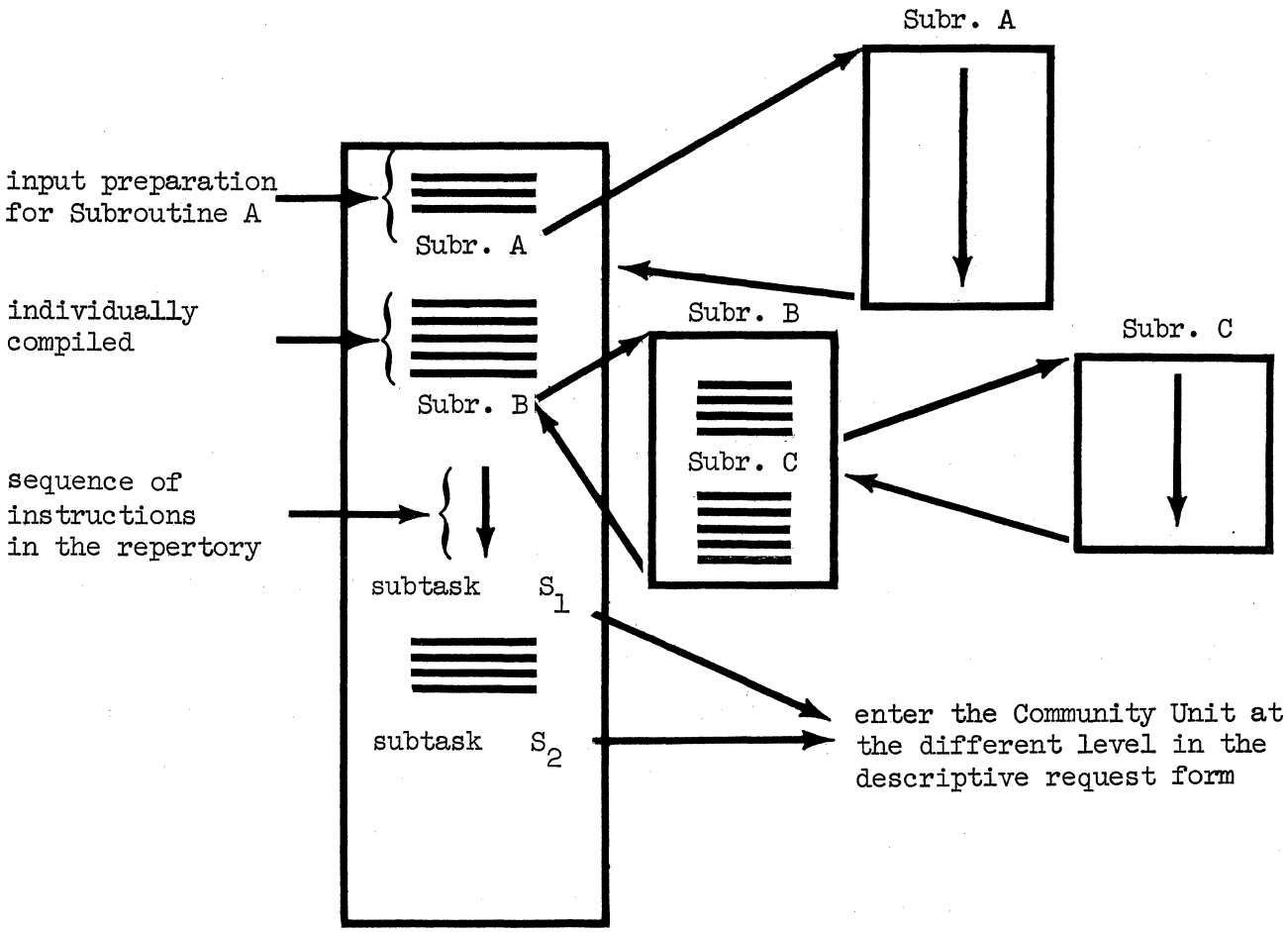


Figure 5.

Monitored and Unmonitored Modes of Execution

instruction, it has no sense of direction toward the goal. This must be provided by feedback from the task analyzer.

Let us review the original request provided in descriptive form. It is comprised of the original state, the desired state, and information on the task. The task analyzer stores all three as the record of the initial task, but it also stores the current state, which is constantly being changed by the executor-monitor. The task analyzer, with the changed picture of the current state will go through the analysis of the changed task for each monitored operation. If the analysis shows the category of the changed task is the same as before, the task analyzer will feedback to the executor-monitor a go-ahead signal so that the executor-monitor will proceed to execute the next operation. If the analysis of the current state changes the category, the task analyzer will feedback to the executor-monitor an interruption signal and then transfer control to the program provider with information about the new problem category. Finally, if the task analyzer is informed of a destructive operation which has been detected by the executor-monitor, an analysis of the error will be made. The information will then be given to the program provider which will make an appropriate modification. In all three cases, the task analyzer keeps the record of changes in the current state and associates the record with the executed instruction. Such records are necessary even for the relatively simple applications described above. In addition it is planned that they will be used by the task analyzer to improve categorization and by the program provider to improve its selection of instructions and programs. Similar changes might be grouped together, abstracted, and generalized. The

resulting summary might be used as a basis for tentatively revising categories and utilty values. The tentatively revised categories and values would be tested and if successful, they would be permanently substituted for the original categories and values.

We have made a number of assumptions about the task analyzer's capabilities and used these assumptions in discussing the ideal functioning of the community unit. Much more research is required before such assumptions can be used with confidence. For example, the learning of good classification techniques is extremely important to a learning system. We can assume that a reasonably rich repertory with good categorization is prestored for the community unit. However, recognizing that new problem situations belong to particular categories in terms of suitable methods requires that pattern-recognition methods of some inductive ability be developed because preprogramming for all conceivable task situations is not feasible. Important research is being done in this area by Minsky (5 and 6), Newell, Shaw and Simon (8 and 9), and Solomonoff (10) among others.

Another serious problem is to provide the mechanism with judgment capabilities comparable to "warmer" and "cooler" feelings of humans. Partial solution to this problem may be possible with a combination of good planning techniques, recognition of partial success, and good credit assigning methods for reinforcement. However, they themselves have many difficulties to be overcome. Discussions of these problems and some suggested solutions are found in (6), (8), and (9).

CHARACTERISTICS AND DETAILS OF THE COMMUNITY UNIT

General features

The structure of the community unit and the function of its members, as discussed in the last section, indicate the extent to which the unit can be preprogrammed. The specification is intended to give the community unit a basic framework with a self-modifying ability which would provide potential capabilities for a variety of tasks. The behavior of the unit at any particular instant, however, is not determined in detail independently of contacts with its environment; its actions are closely guided by feedback from the environment.

The structure presented in the schematic diagram of Figure 3, is a "building-block" structure to be found in several other parts of the system, e.g., in the planning mechanism and in the induction mechanism. The interactions between the executor-monitor and the task analyzer represented by opposing arrows in Figure 3, form what we call a first-order feedback loop; its primary function is performance and error detection. This loop resembles the TOTE unit in Figure 4, with the executor-monitor corresponding to the operate phase of the TOTE unit and the task analyzer corresponding to the test phase.

The outer loop which connects all three members of the community unit may be called a second-order feedback loop; its primary function is the selection of operations and error correction on the basis of the information from the first-order feedback. For example, in connection with the hammering task represented by the TOTE unit, what happens if the hammerer hits his thumb instead of the nail? Something must be changed to correct the situation

so that the "wrong" action will be prevented from recurring and a correct action substituted for it. This correcting process, after a unit of action has taken place, is accomplished by the second-order feedback, with or without additional feedback from the environment.

In order to compare the structure of the community unit with that of the TOTE unit, the selection of operations can be considered as a part of an operation phase, as when Miller et al. talk about "metaplanning"--about TOTE units which construct other TOTE units--but for our purposes it is convenient to schematize selection of operations as separate from performance of operations.

Hierarchical structure and evaluation of accomplished tasks

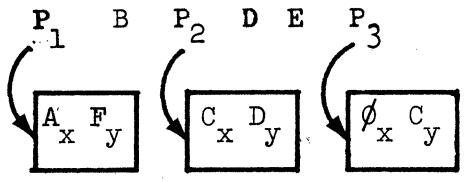
Whenever a request is made of the community unit, the only criteria which the unit can use in determining acceptability of a response are those determining whether the response satisfies the request. However, the request itself, dependent as it is on heuristics used by a higher-level program which generated it, may have been inadequate. Depending on an evaluation made at the higher level, the particular program produced by the program provider may have to be modified. Such a requirement will be fed back to the unit. The task analyzer then may decide to adjust the categorization, and subsequent adjustment of the utility values may have to be made by the program provider. Full control must not be passed down. The request must be associated with some "resource allotment" or other constraint to help it recognize when help is needed.

Abstraction and generalization

Much more thought is necessary to decide how to equip our system with abstraction and generalization capabilities. This section supplies some preliminary suggestions, referring first to data, then to programs. Uses to which the abstracts may be put will be discussed later.

Given a list of lists, the abstraction routine prepares three kinds of data abstracts. The simplest is obtained by counting the elements in each list. Another type, taking order of occurrence into account, itemizes what elements the lists in the list have in common, and replaces the unmatched elements by a sublist which indicates how the parameter is specified in each of the original lists. The third type of data abstract, without taking order of occurrence into account, itemizes the distinct elements which appear in common among the lists, then itemizes the unmatched elements together with names of the original lists to which they belong.

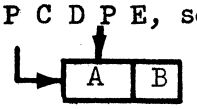
For an example of the three types, let two lists X and Y, be given to the abstraction routine. List X contains elements A B C D E and list Y contains elements F B D D E C. The abstract which counts data elements has as entries 5_x, 6_y. Subscripts x and y are used to indicate names of the original. The abstract taking order into account may be represented as



where P_1, P_2, and P_3 are parameters, each pointing to a specification sublist. Letting "/" symbolize a marker separating shared elements from

unshared ones, the third type of abstract is represented as B C D E / A_x F_y.

Another abstraction routine, called a "replace routine", replaces one or more specified elements in a given list by a single element, using either a parameter supplied by the routine or a particular name given by the controlling program. The original element(s) then form a sublist which is named by the new element. For example, suppose the higher-level program enters the routine with the information that in the list Z, which contains elements A B C D A B E, elements A and B together are to be replaced by a parameter. The resulting list is P C D P E, so that A and B are treated as



a single element at the level of list Z, but become individual entities again at the next lower level.

Programs may also be abstracted and generalized. A subroutine, when first constructed or when prestored in the community unit, may not be in a form suitable for general use. After a suitable training sequence, the unit can be made to discover that some subroutines have many common instructions and only a few differences. It then replaces these subroutines by one subroutine which contains parameters at places where differences appear. Differences may be in operators (instructions), in operands (addresses), or both. This recognition of common features is a form of both abstraction and generalization. For the former, when the program provider decides to use an abstracted subroutine, it will have to copy the subroutine and specify values of parameters before the subroutine can be given to the executor-monitor. When an abstracted routine has proved its power by frequent use, it may be

generalized as a closed subroutine. Generalization is then said to occur. The name of the subroutine is stored together with the necessary format for the calling sequence, and the program provider utilizes the generalized subroutine by putting in its generated programs instructions which specify values of the calling parameters and which then transfer control directly to the stored subroutines. Abstraction and generalization can happen to routines and subroutines in both the temporary set and the standard repertory but they are more likely to happen to those in the temporary set.

Transfer of subroutines from the temporary set to the repertory

If a subroutine in the temporary set is used successfully some preset number of times without modification, the program will be made into a closed subroutine and its name will be stored in the standard repertory. This enables well-tested and often-used programs to be executed in unmonitored high-speed by the executor-monitor.

Analogies between a human and the community unit

Some interesting analogies can be drawn between learning processes of the human and those of the community unit. One is the development of high-speed performance following sufficient monitored experience. When a sequence of actions with which we are unfamiliar is first proposed (either by our teacher or by ourselves) we consciously attend to each step of the sequence. However, once we gain familiarity and confidence, we run through a sequence of actions without being consciously aware of each of its parts. The difference can be seen in the rapid and precise finger movements of a skilled piano player compared with the slow trial-and-error movements of a beginner.

This corresponds to monitoring a program step-by-step when the community unit is unsure of workability of the program, but changing to high-speed, unmonitored execution once the effects of a program have been thoroughly tested.

Another similarity occurs in the progressive grouping of different elements into larger and larger complexes. Suppose we have acquired skills in a number of simple tasks (e.g., some sequences of finger movements on a piano). When we attempt a more complex task which is an integrated sequence of those simpler skills, we have to attend to each step of the sequence although we need not attend to the full detail of each of the basic skills. Some parts of a basic skill may have to be modified to fit it into the larger sequence, and we are not sure of the effects of the basic skills on each other when they are combined. But with practice these questions are settled, and the basic, simpler skills drop out of our attention as individual units, forming a fused integrated whole. The process can be repeated any number of times, forming larger and larger units of behavior, the ultimate size of units depending on the proficiency we acquire in a particular field of skills. The community unit functions in the same way, forming larger and more complex routines out of simpler subroutines.

Still another similarity between human and community unit experience is a tendency toward abstraction as experience builds up. In many motor skills, we observe that some component skills are very specialized and fit rigidly into a larger pattern. Some component skills, however, when used in a situation calling for variations, can quickly be adapted. Of course,

the opposite effect may be evidenced, i.e., previous acquisition of a skill may interfere with the learning of a new task. After we experience a number of variations of the same basic skill, we generally find it easier and easier to adapt the skill to still different situations. An analogous behavior in the community unit results from abstraction and generalization of subroutines. A subroutine can be made to handle a variety of tasks by means of parameterization; it is adapted to a new situation by determining new values for its parameters.

Generalized request form

Descriptive request forms used thus far are rather limited in their capability to express a wide variety of problems. In order to reduce the limitation, we consider a more general form which includes the previous form as a special case. This general form is represented by three lists of information:

(1) given: facts and conditions concerning initial situation.

(2) desired: end results to be obtained.

(3) information on the task: restrictions and suggestions.

Corresponding generalizations of the community unit function and its cognitive map construction have not been fully investigated.

Category changes during execution of the program

As the task analyzer receives executed results from the executor-monitor the current state becomes altered from the original state, although the desired state is the same. Some changes in the current state may be very small or very large; if the nonmonitored part of the program is a large

routine in the repertory, changes noticed by the task analyzer in the current state are likely to be large since the executor-monitor cannot regain its control until that portion of the program is executed. In such case, the task analyzer may find the new current state to be in a different category. On the other hand, if individual instructions are monitored, the resulting change in the current state will be small.

It has been found that looking at the current task as ever changing, in the fashion described in the preceding paragraph, has a great advantage. However, it has its pitfalls. If the categorization is poor, and if the program provider has to construct a complex program from the basic instructions, it may never find a fruitful path since the task analyzer may stick to the wrong category because of small changes reported by the executor-monitor.

In order to remedy this situation, planning is needed to guide the course of action, i.e., to place guideposts on the road to the goal. The next section discusses such possibility.

PLANNING: GUIDEPOSTS ON THE ROAD TO THE GOAL

Minsky (6) points out, "practically any ability at all to 'plan,' or 'analyze,' a problem will be profitable, if the problem is difficult." To illustrate the point, he says, "Generally 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!" Additional discussions and useful suggestions in this area are found in the references (5), (6), (8), and (9).

Proposed mechanism

The procedures described below have many resemblances to the procedures proposed by Newell, Shaw, and Simon for their General Problem Solver. See (8). Our planning mechanism (Figure 6) is similar in structure to the community unit. In fact, the planning mechanism uses, in addition to its own records, the same record in the memory which the community unit uses. We again assume that requested tasks are in descriptive form.

We propose to use a set of characterizing expressions 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 operattions 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 expressed 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 set of methods or operations will be; descriptions

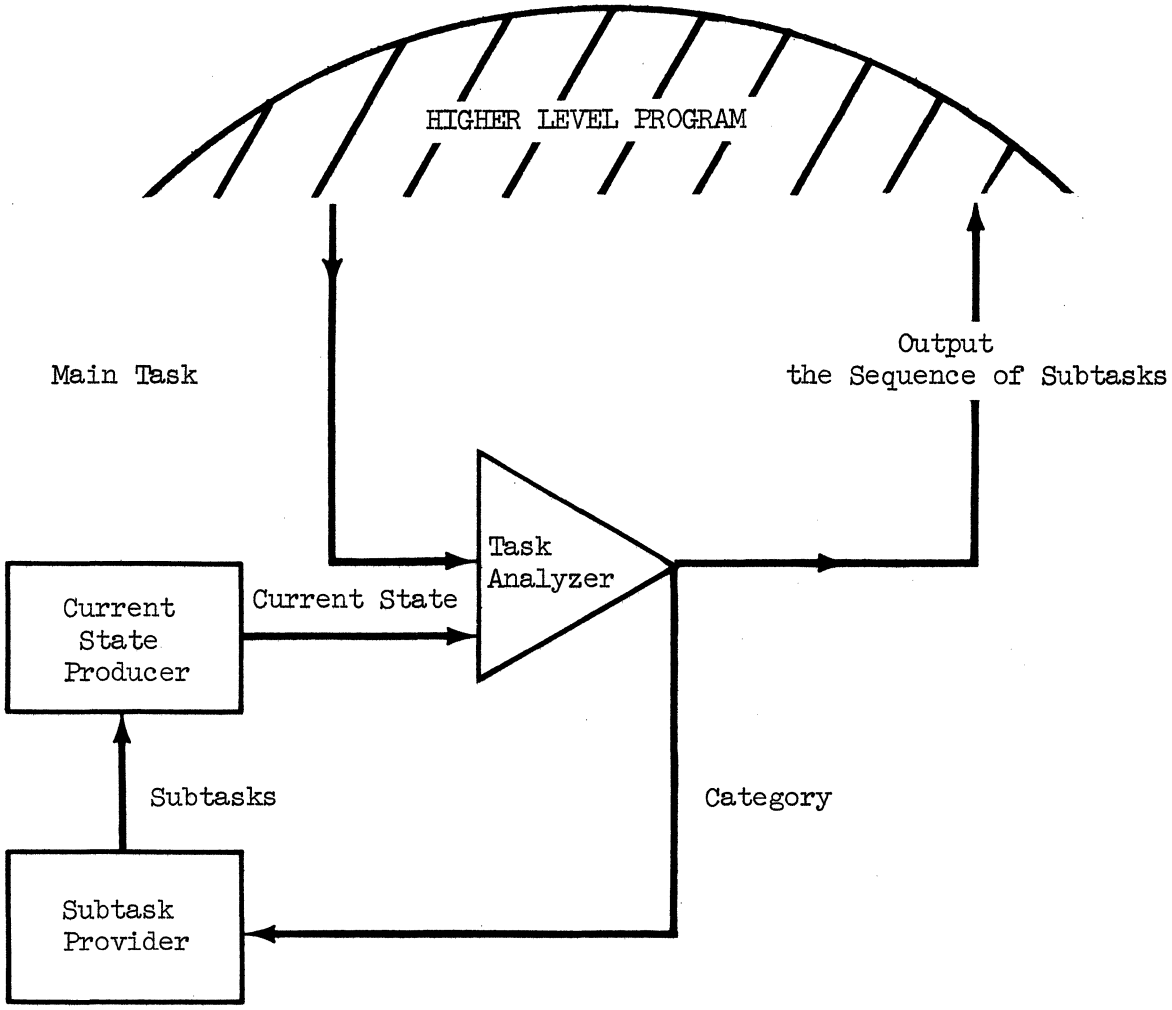
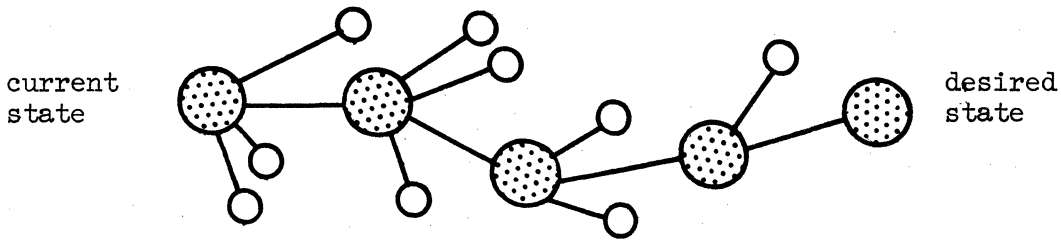


Figure 6.

Planning Mechanism

of input and output will be only in general terms 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, 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. Newell, Shaw, and Simon's GPS.) Given the new task, defined by the chosen new current state and the originally given desired state, the same sequence of

steps is repeated.

Shaded nodes of the graph indicate grouping of "similar" states using coarse task categorization during the early planning stages. A rough plan is developed using a small set of coarse categories; then subplans connecting the guideposts (nodes on the graph) and using finer categories are developed; then subsubplans are developed using still finer categories, etc. 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 in their turn divided into smaller subtasks by the same mechanism, similarity criteria are tightened and there must be a more and more exact match of states.

It should not be overlooked that although the process of performing the 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. It should also be observed that the system may propose many plans. One plan, which seemed plausible at its early planning stage, may be found to be unworkable as finer and finer details are supplied later at a more concrete level. Then the categories will be coarsened and the criteria relaxed again, and a new plan will be formulated either by modifying the old one or beginning again.

An illustration

Part of the general strategy used for the "Tower of Hanoi" puzzle is illustrated in Figures 7 and 8. A description of the puzzle is provided later, but it is not necessary at this point to understand the puzzle. It is necessary only to follow the diagrammatic representation of tasks and subtasks in the figures.

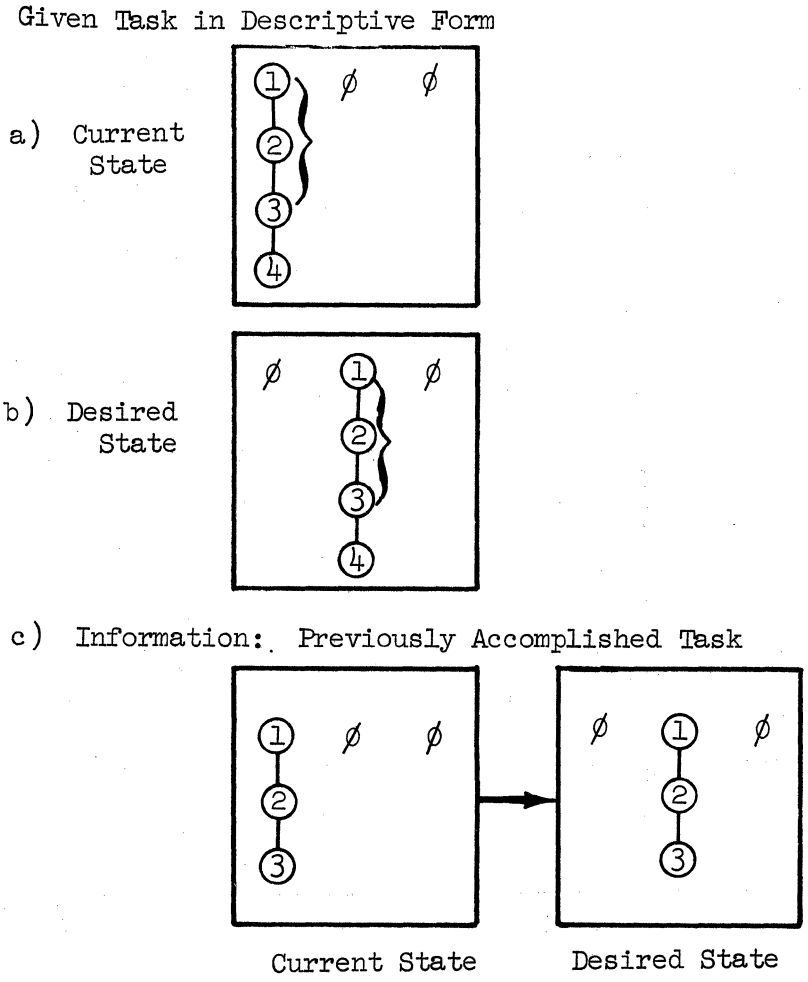


Figure 7.

Tower of Hanoi Puzzle Given to the Planning Mechanism

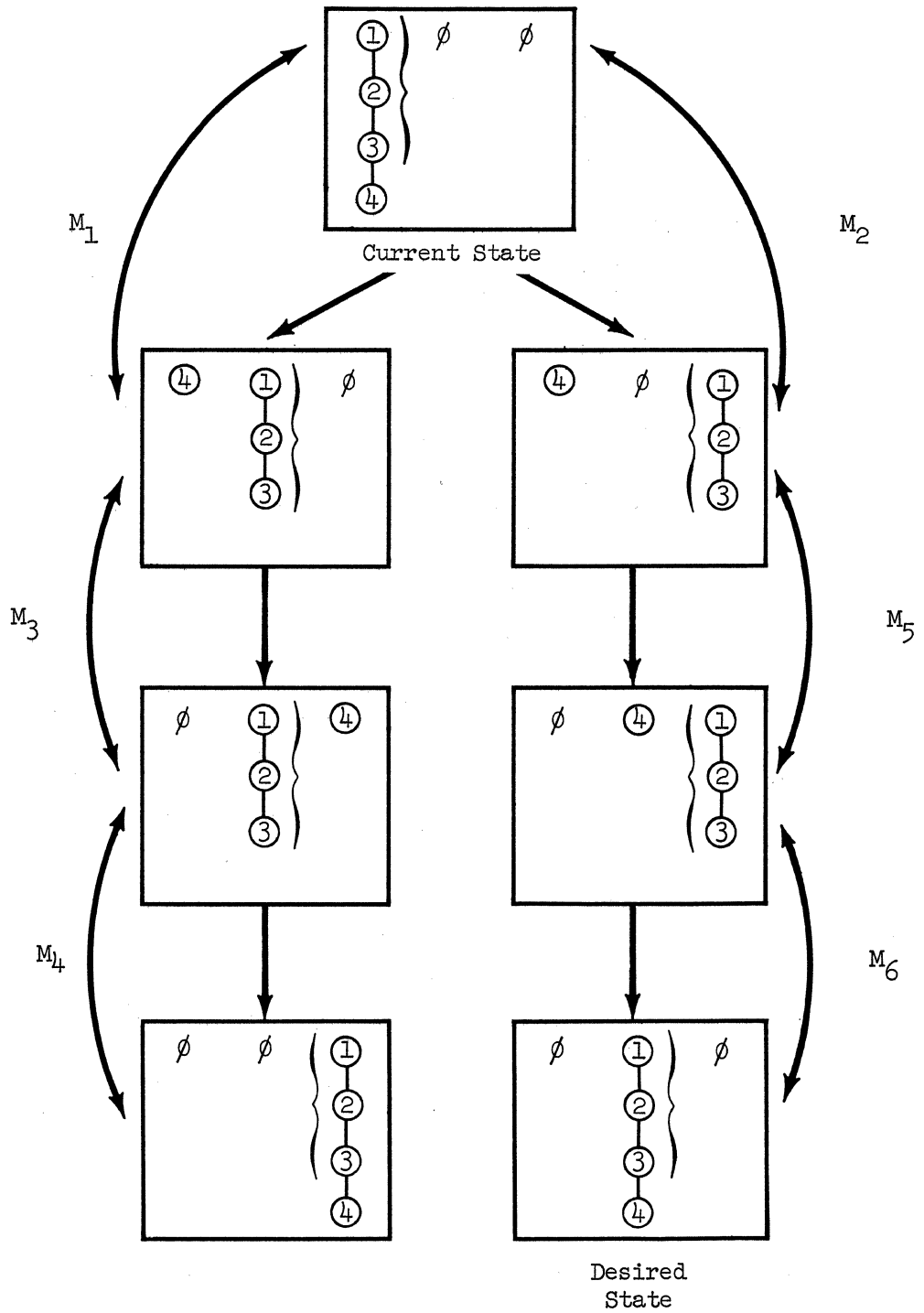
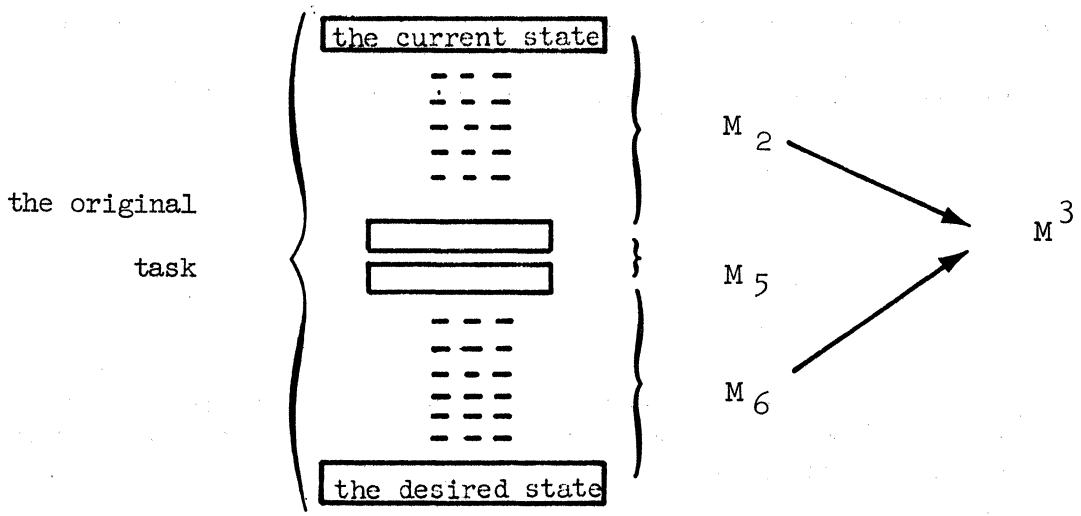


Figure 8. Subtasks of Tower of Hanoi Puzzle

Figure 7 is a schematic representation of a task given in descriptive form. The task analyzer uses the information given about the previously accomplished task in the following way: it uses its abstraction routine to find that the given task has one more element (4) than the previously accomplished task, and it uses its replace routine to combine (1), (2), and (3) into one element.

The subtask provider, using a legal move generator, produces both M_1 and M_2 as two possible legal moves. Each M_i is stored by the program in descriptive request forms. From these the current-state producer finds and outputs the desired states to the task analyzer as current states of new subtasks. The task analyzer chooses 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 below.



The original task is represented by the space from the top to the bottom rectangles, 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[1] 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; 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 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

---

[1] 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[1] before it finds the right sequence. This is an exhaustive search. Planning of this kind is relatively cheap--it takes six times around the loop 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 would take 65,534 examinations.

We have not considered here the "difficulty estimate" of each proposed subtask (see Minsky (5)). However, such information, both given and based on experience, must be incorporated in the decision-making criteria used by the task analyzer.

AN ILLUSTRATION OF COMMUNITY UNIT OPERATION

Consider a fairly simple programming task, to exchange two symbols A and B, currently stored in locations L_1 and L_2 respectively, so that A is moved to L_2 and B to L_1. The task is given to the community unit as a descriptive request form expressed as a list with current state: A B ; desired state: B A ; and information on the task: a code indicating that the desired state is to replace the current state.

Task analyzer's work

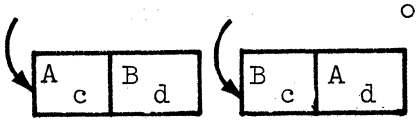
The task analyzer, in an attempt to find a suitable category for this task uses the abstraction routine discussed previously to compare the current and the desired states. The following three abstracts result:

---

[1] 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 Figure 8). 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.

2_c, 2_d (meaning both the current and desired state lists contain two elements).

(P_c, P_d) (meaning there is no common element when order of occurrence is taken into account).



(A, B) (meaning exact match in content when the order is immaterial).

The third abstract indicates that no element is added or deleted, while the second and third indicate that the required operation involves moving contents of cells without changing the contents. From these conditions, the task analyzer determines an appropriate problem category.

The program provider's work

Let us assume that the category determined by the task analyzer already has associated with it a set of instructions and sequences of instructions. We also assume, for simplicity, that the instruction pair CAD and STO has the highest utility value of the pairs whose description matches the information deduced from the second and third abstracts. Instruction "CAD L" means "clear and add (load) into the accumulator (ACC) the content of memory register L," and instruction "STO L" means "store the content of ACC in the memory register L."

The program provider might produce CAD L_1, STO L_2 as a tentative sequence of operations, simply because L_1 and L_2 are the only locations involved in the request. This sequence is given to the executor-monitor, with a flag indicating it is to be monitored rather than executed at high speed.

Interactions of members

Figure 9 shows in tabular form the action of each member of the community unit. Arrows indicate the sequence of control. From the program provider on one line, control always goes to the executor-monitor on the next line. Interaction between the executor-monitor and task analyzer is indicated by opposed arrows. When no entry is indicated on a line for the "Current Program in Storage" column, the entry is assumed to be the same as for the previous line. The column on the extreme left indicates the current operation under consideration by the executor-monitor. Some trial and error actions are not explicitly indicated here but are to be understood whenever the program provider modifies or adds instructions. Many modifications of the initial modification may be necessary.

Our attention is now on the executor-monitor. Upon receiving two instructions with a monitor flag, the executor-monitor places danger tags on all addresses involved. When CAD L_1 is considered, it detects no danger since ACC, in this case a particular cell which is set aside to serve as a mock accumulator, has no tag attached. The information (ACC) = A, where "(ACC)" designates the content of ACC, is given to the task analyzer, which modifies the current state and returns control to the executor-monitor with a go-ahead signal. The executor-monitor now places a danger tag on ACC and executes CAD L_1 interpretively by storing the content of L_1 and ACC.

The next instruction to be considered is STO L_2. Since L_2 had a tag, the interruption is recorded and the executor-monitor outputs a danger signal to the task analyzer. The task analyzer determines that the content of L_2 in the

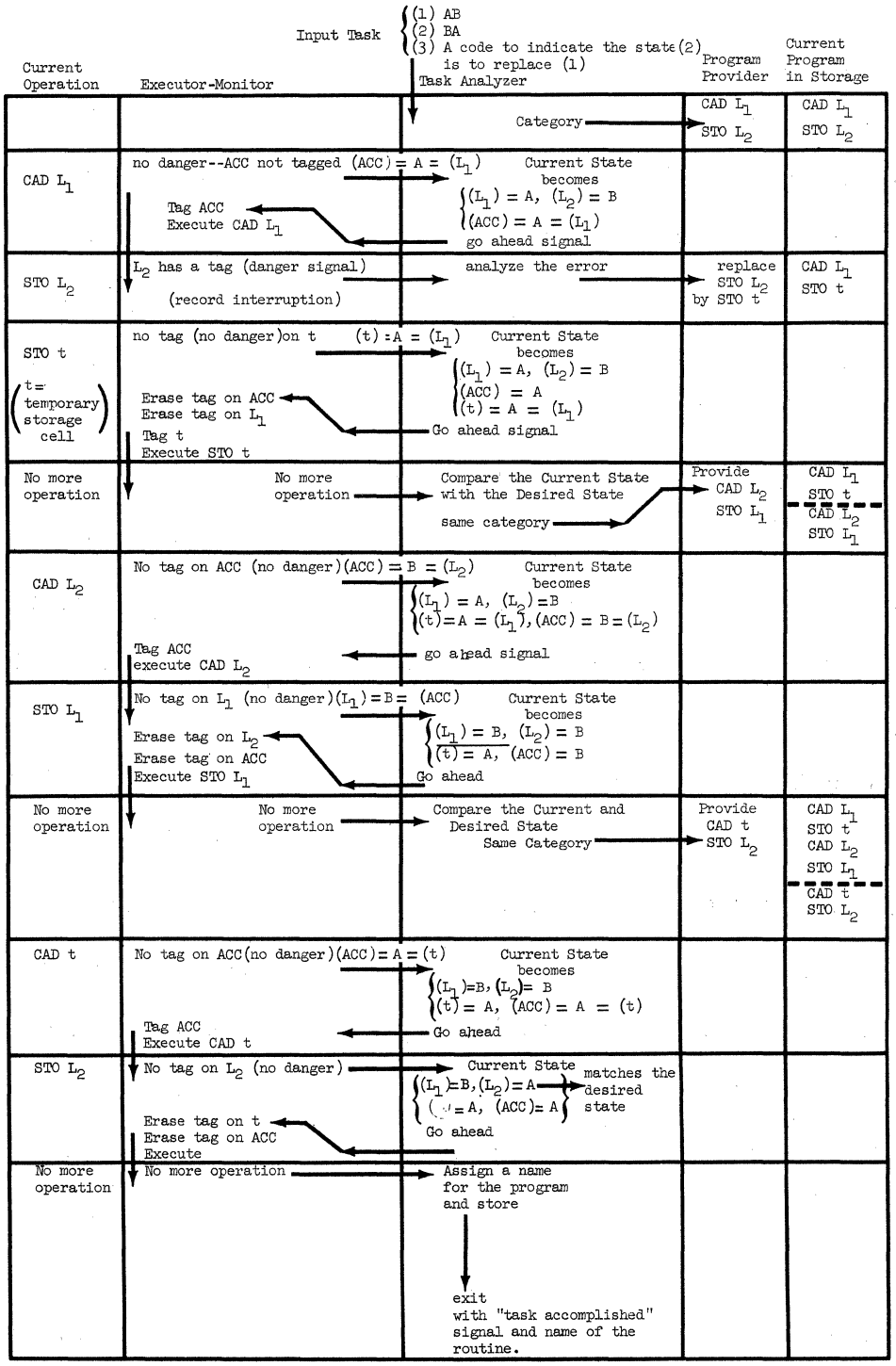


Figure 9. Actions of Members of the Community Unit

current state is not to be destroyed and gives the category of the particular error to the program provider. The program provider replaces the STO L_2 by STO t (t being any temporary storage location) and transfers control to the executor-monitor.

The executor-monitor now considers STO t, detects no danger, and outputs the information (t) = A = (L_1) to the task analyzer. Similar processes occur, as indicated in Figure 9, until the task analyzer finds the current state to be the same as the desired state. It then assigns a name to the produced routine, stores name and routine, and exits to the higher-level program with a "task accomplished" signal and the name of the routine.

When the same request is made again, the request in either appellative form or descriptive form will cause the program provider to output the same routine to the executor-monitor. However, the routine will continue to be monitored until a preset number of successful executions of the routine have been recorded.

When the community unit has experienced similar requests, such as

C D in L_3 L_4 location	E F in L_5 L_6 location
D C	F E
same as before	same as before

the abstraction routine for programs can produce a routine like this:

```
CAD P1
STO t
CAD P2
STO P1
CAD t
STO P2
```

together with an abstracted request form $\begin{cases} (P_1), (P_2) \\ (P_2), (P_1) \\ \text{name of abstracted routine} \end{cases}$

where "P_1" and "P_2" indicate parameterized addresses and "(P_1)" and "(P_2)" indicate contents of these addresses. This permits the community unit to handle binary exchange requests using any addresses.

TOWER OF HANOI PUZZLE

The "Tower of Hanoi,"[1] illustrated in Figure 10, was invented by a French mathematician and sold as a toy in 1833. The problem 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 $2^n - 1$. Thus three disks can be transferred in seven moves, four in 15, five in 31 and so on. For eight disks, the usual number considered for the toy, 255 moves are required.

In the original description, the toy is described as a simplified version of a mythical "Tower of Brahma" in a temple in the Indian city of Benares. This tower, the description reads, consists of 64 disks of gold in the process of being transferred to another needle, by the temple priests. When the transfer is completed, the temple is expected to crumble into dust and the world to

---

[1] Gardner, Martin, Mathematical Puzzles & Diversions, pp. 57-59.

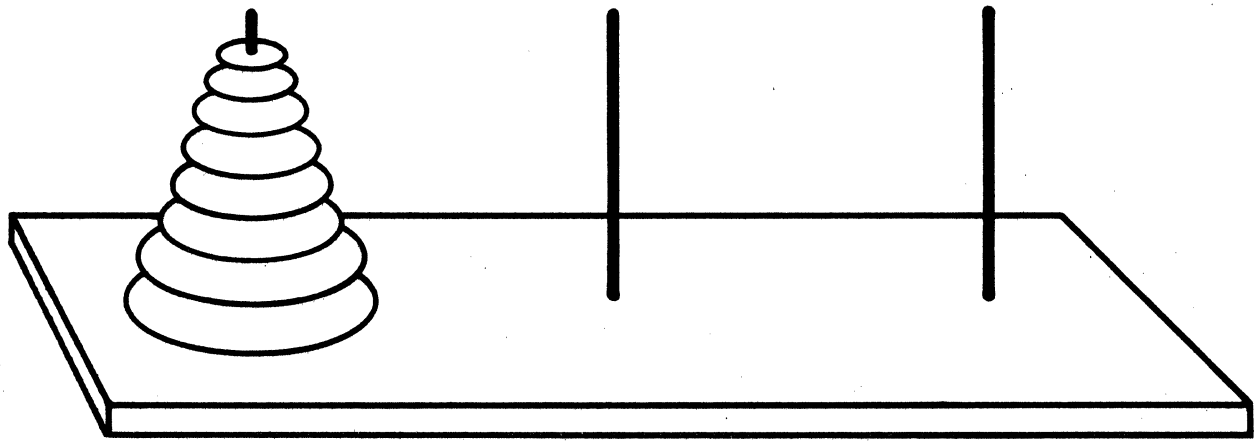


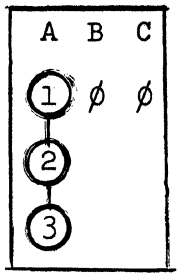
Figure 10.
The Tower of Hanoi Puzzle

vanish in a clap of thunder. The disappearance of the world may be questioned, but there is little doubt about the crumbling of the temple. $2^{64} - 1$ is the 20-digit number 18,466,744,073,709,551,615. Assuming that the priests work night and day, moving one disk every second, and assuming that the priests know the shortest sequence of moves, it will take them about 585 billion years to finish the job.

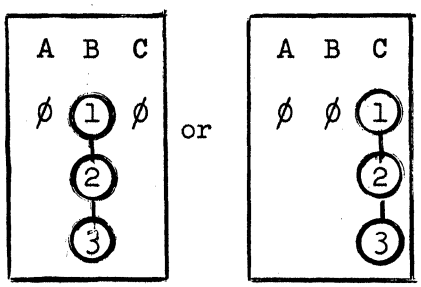
This puzzle was chosen as the first testing vehicle for our system for several reasons. Its solution is relatively simple but not trivial. Results with human subjects cover a wide range; when the task was stated as "Find the shortest sequence of moves for eight disks and find a general principle which uniquely determines this sequence," the time required for solution ranged from minutes to days. For some it was unsolvable. Another reason is that the solution is known to the 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 use only 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. The following illustration shows the task for the three-disk case.

Current state:



Desired state:



Information on the task: Rules of the game were presented in the form of a program which generates legal moves when the current state is given. It is possible, however, to present the rules of the puzzle in a descriptive form and to let the system produce its own program for legal move generation provided that such a description uses terms and formats interpretable by the system.

Columns A, B, and C in the request form diagrams above represent the three pegs in the puzzle and circled (1), (2), and (3) are numbered disks from the smallest to the largest. The symbol ∅ 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 11 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 Figure 11).

How many such nodes (moves and current-state configurations) exist for an exhaustive search? Since there are two branches from every node, there are 2^m possibilities at the m^{th} level (counting from the top in descending the tree). In order to reach from the top to the m^{th} level by the exhaustive method, there are $\sum_{i=1}^{m} 2^i$ current states to be examined. Since the smallest number of moves for n disks is known to be $2^n - 1$, there are $2^n - 1$ levels to consider before the puzzle solver arrives at the desired state. Therefore, 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.

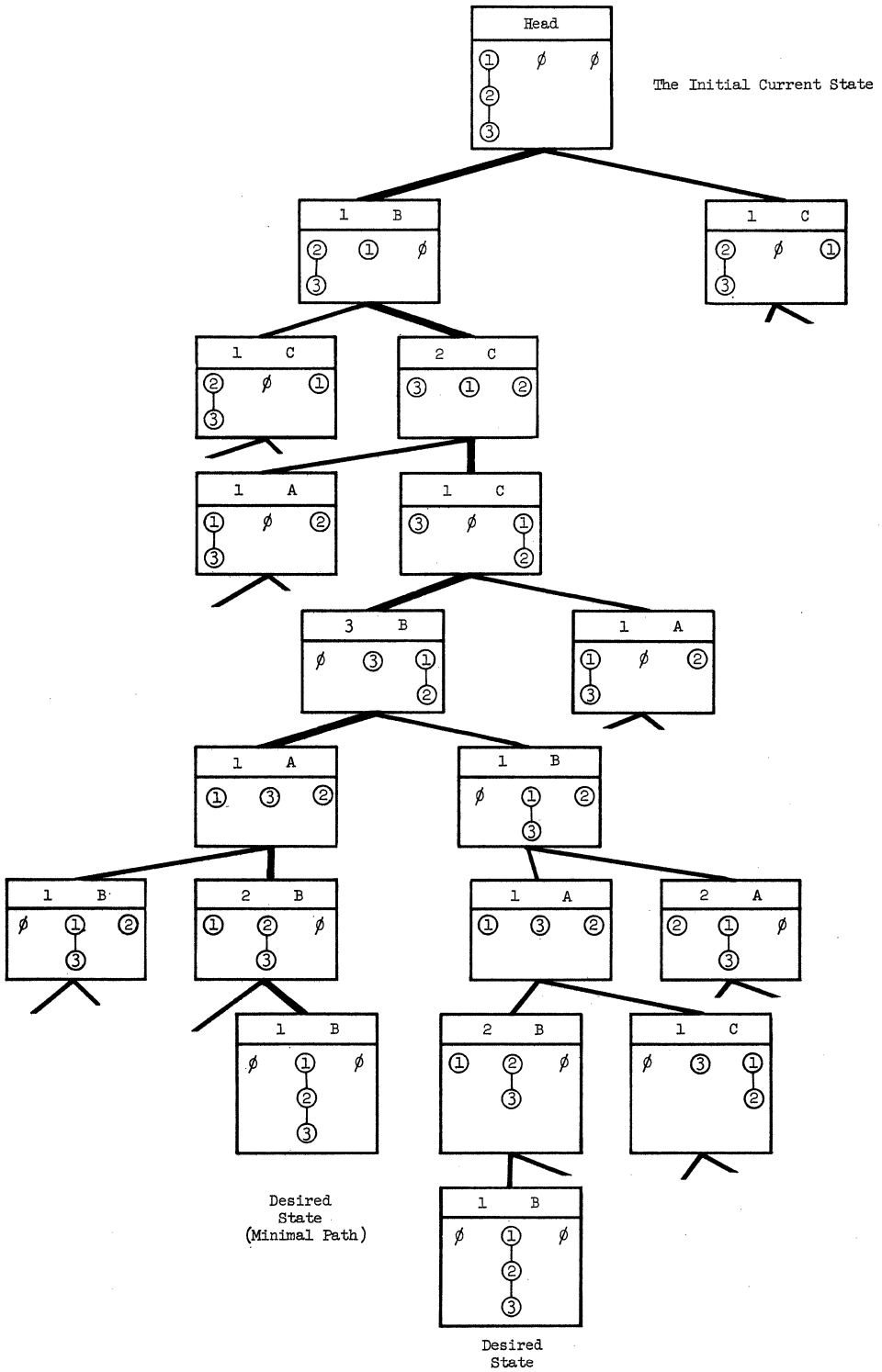


Figure 11. Part of Move Tree for the 3-Disk Puzzle

To calculate for the seven-disk case, take the last number plus 2, which is 2^{2^6}, square it to obtain 2^{2^7} and then substract 2 from it. The result will be a 39 digit number. A 78 digit number will result for the eight-disk case.

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., Kochen (2), Solomonoff (10 and 11), and Watanabe (13).

The inductive procedure observed in humans may be typified and described in general terms: when we want to formulate general rules about a class of phenomena, we first make a guess to form a hypothesis; next we deduce certain consequences of the hypothesis and test them against new evidence and old; and then we increase our confidence in the hypothesis, modify the hypothesis, or form a new hypothesis and repeat the procedure.

For our experiment, we give to the proposed system, as a training sequence, inductive tasks of a simple form. A set of general rules to be formulated by the system is unique and known to the trainer 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 previously formed hypothesis or the forming of a new hypothesis by the system.

The mechanism proposed has a structure similar to that of the community 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 the claims by some, 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 (1).

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

Suppose the system has found the successful sequence of moves for the three-disk case, i.e., 1 2 1 3 1 2 1, where the top line shows particular
B C C B A B B
disks moved and the bottom line shows the names of columns to which these disks are moved. For instance, 1 means that the disk (1) was moved to
B
column B.

The four-disk case is now given, and the system is asked to find the successful sequence for the new case. The subsystem, which we call the induction mechanism, then goes to work (see Figure 12).

Observation and analysis

The first phase of the inductive process is performed by the task analyzer. For the Tower of Hanoi puzzle, the task analyzer begins by comparing the descriptive request form of the three-disk and four-disk cases by means of the abstraction routine (see Figure 13). The conclusion is that both cases are identical except that the four-disk case contains an additional disk,

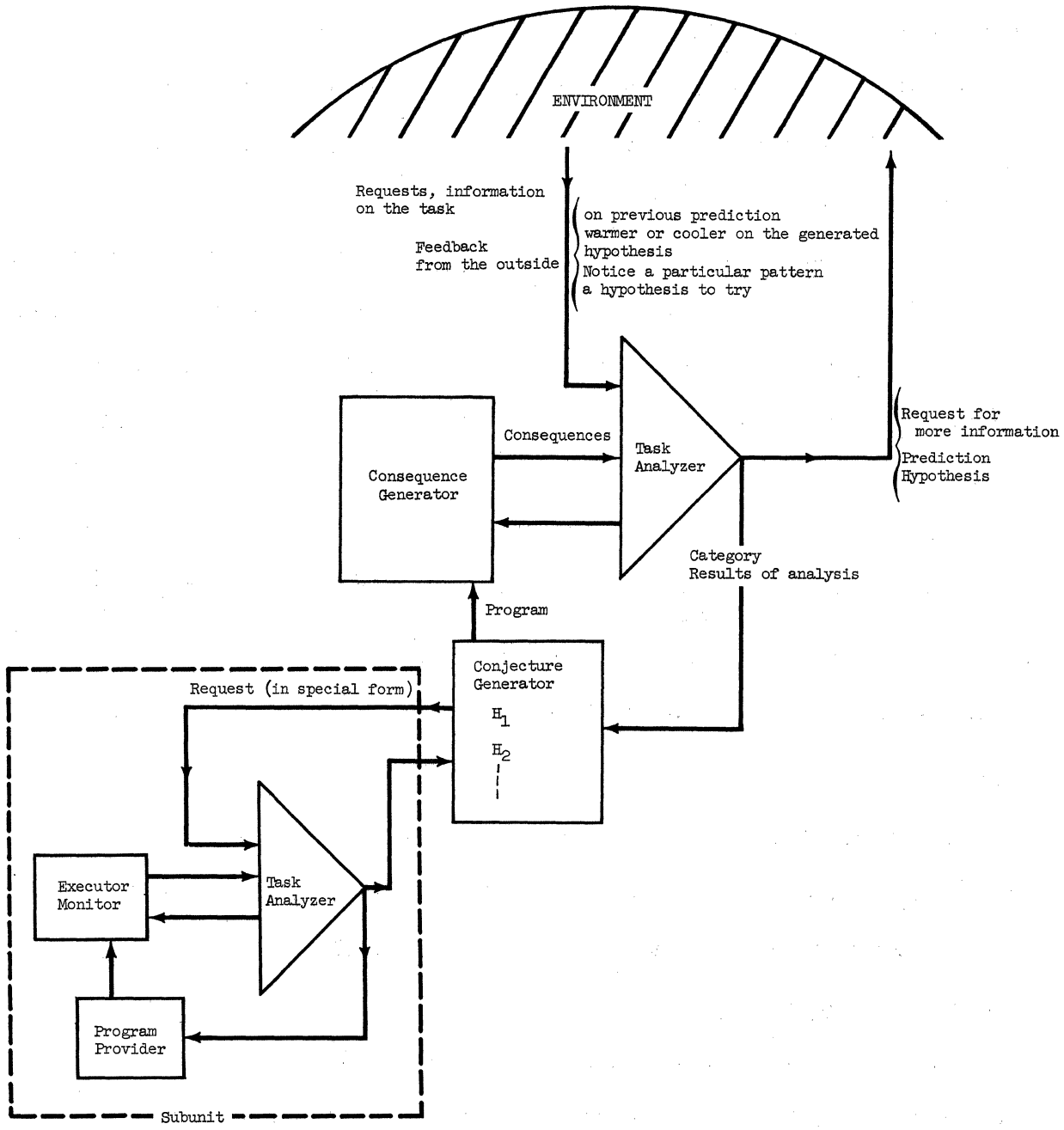


Figure 12. Induction Mechanism

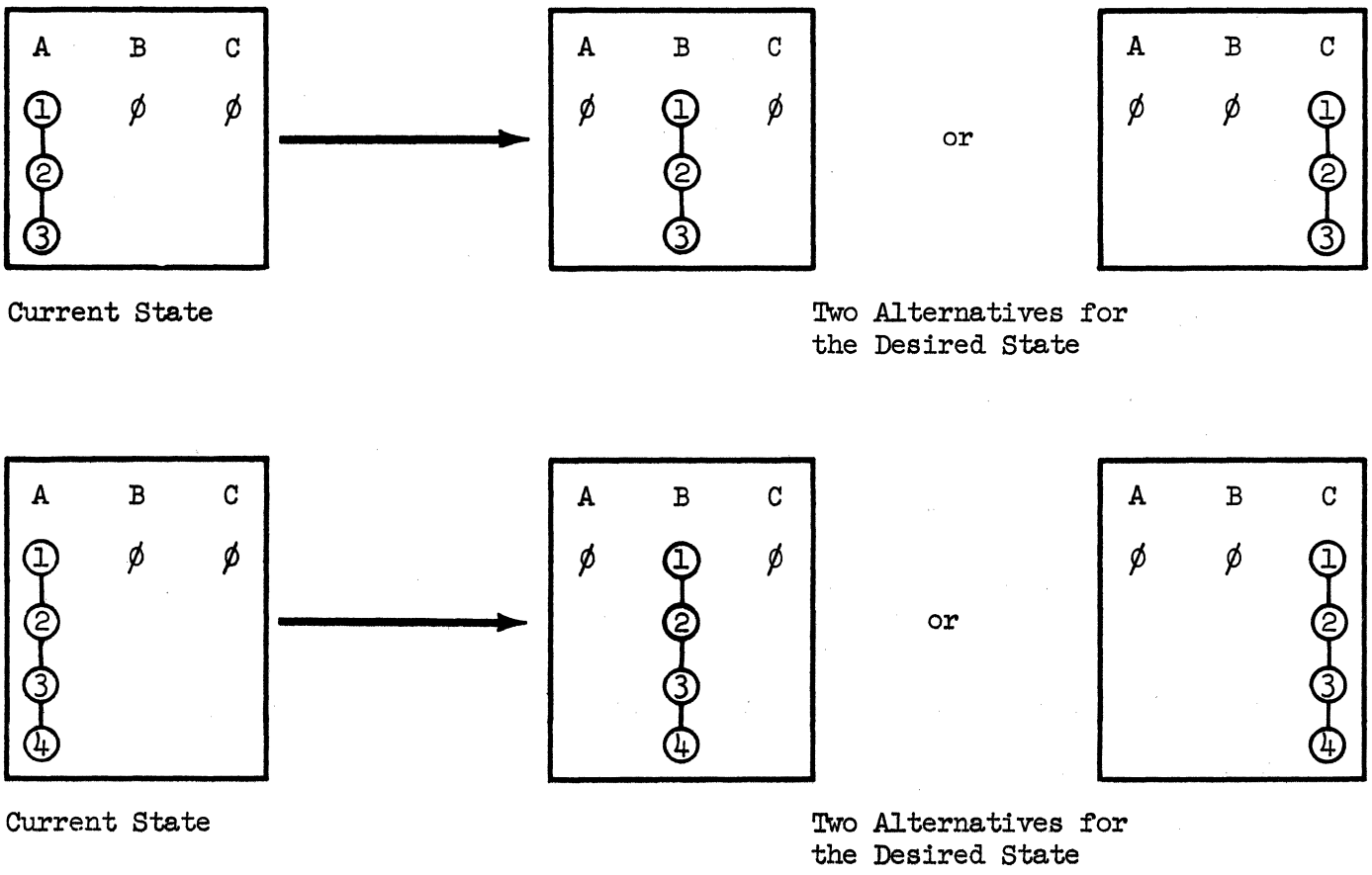


Figure 13.

3 and 4-Disk Puzzles

disk (4), in both the current and desired states. For the next step the successful sequence of moves for the three-disk case and the elements appearing in the description of the case are compared with the abstraction routine. The conclusion is that the elements are the same, i.e., that both contain A, B, C, (1), (2), and (3) and nothing else. The conjecture is made that if the four-disk case is to follow the same pattern, the sequence of moves must contain the additional element (4). Then within the three-disk sequence, it is discovered that some elements are repeated. The task analyzer now outputs the results of its analysis and a problem 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 Figure 12). 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 three lists of information:

(1) given: a sequence

(2) desired: a program to regenerate the sequence

(3) information on the task: restrictions and suggestions (characterization of the sequence from the task analyzer of the induction mechanism).

In our example, the conjecture generator receives "cyclic" as the problem category together with the sequence of successful moves for the three-disk case, i.e., 1 2 1 3 1 2 1. The conjecture generator separates the top from the
B C C B A B B

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[1] and gives it to the executor-monitor:

L_x	CAD L_1	"L_1" is the address of where the symbol (1) is stored
	TR L_y	"TR" is an abbreviation for "transfer control to"
	CAD L_2	"L_2" is the address of where the symbol (2) is stored
	TR L_y	
	TR L_x	"L_x" is the address where the first instruction of the sequence is stored.
		"L_y" is the entering address of a special program in the consequence generator which manipulates, examines and uses the content of the accumulator each time the entrance is made, and returns control to the next instruction in the proposed sequence. However, while the proposed program is being monitored, "TR L_y" will not cause an actual transfer to L_y; instead the task analyzer will note the location L_y and instruct the executor-monitor to proceed to the next instruction.

[1] The program is written in IPL-V language but is translated into CAD's and TR's here for mnemonic purposes.

Let us examine the interaction between the executor-monitor and the task analyzer when these instructions are monitored. The proposed program brings the symbol (1) and the symbol (2) to the accumulator alternately. When the task analyzer receives information on the content of the accumulator from the executor-monitor, it compares the content with the appropriate element in the given sequence. 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 item "1 2 1 3" as defining a cycle phase. This time the program provider constructs a program (the location L_3 contains the symbol (3)):

```
Lx  CAD  L1
    TR   Ly
    CAD  L2
    TR   Ly
    CAD  L1
    TR   Ly
    CAD  L3
    TR   Ly
    TR   Lx
```

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 Figure 12).

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 phase.

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 human being or the higher-level programs serving as the environment of the mechanism determine the success of the predictions and can either give to the mechanism some additional information on the task or indicate to the mechanism whether its prediction was successful or not. The additional information might be a particular hypothesis (in program form) to be tried out, or it might be an indication of what a prediction should have been. The task analyzer uses such feedback from the environment, in addition to what was previously given, as a basis for a new analysis, and the entire process is repeated.

Let us return to our specific example. The two programs which have

been constructed by the conjecture generator are executed by the consequence generator to produce suggested moves for the four-disk case. The task analyzer proposes these to its higher-level program. The higher-level program 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, it is output to a human teacher who examines it and feeds back whether the move is right or wrong. Such feedback is relayed to the task analyzer.

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.

Suppose the higher-level program finds correct moves, by some trial and error method or by asking the teacher, and informs the task anlyzer. The task analyzer now determines the unmatched elements in the suggested sequence and the correct-move sequence and informs the conjecture generator. The conjecture generator modifies previously constructed programs so that they will replace unmatched places with parameters. The resulting programs, when executed would produce a sequence like this:

$$\left\{\begin{array}{l} \underline{1\ 2\ 1\ P_1\ 1\ 2\ 1\ P_1\ 1\ 2} \ - - - - \\ \underline{B\ C\ C\ P_2\ A\ P_3\ B\ C\ C\ P_2} \ - - - - \end{array}\right.$$

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:

$$\left\{\begin{array}{l} 1\ 2\ 1\ P_1\ 1\ 2\ 1\ P_1\ 1\ 2\ 1\ P_1\ 1\ 2\ 1\ P_1\ 1\ 2\ 1\ P_1\ 1\ 2\ 1\ P_1\ 1\ 2\ 1\ P_1\ 1\ 2\ 1 \\ B\ C\ C\ P_2\ A\ P_3\ B\ C\ C\ P_2\ A\ P_3\ B\ C\ C\ P_2\ A\ P_3\ B\ \boxed{C}\ C\ P_2\ A\ P_3\ B\ C\ C\ P_2\ A\ P_3\ B \end{array}\right.$$

correct moves:

$$\left\{\begin{array}{l} 1\ 2\ 1\ 3\ 1\ 2\ 1\ 4\ 1\ 2\ 1\ 3\ 1\ 2\ 1\ 5\ 1\ 2\ 1\ 3\ 1\ 2\ 1\ 4\ 1\ 2\ 1\ 3\ 1\ 2\ 1 \\ B\ C\ C\ B\ A\ B\ B\ C\ C\ A\ A\ C\ B\ C\ C\ B\ A\ B\ B\ A\ C\ A\ A\ B\ B\ C\ C\ B\ A\ B\ B \end{array}\right.$$

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. Among the suggested moves, 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:

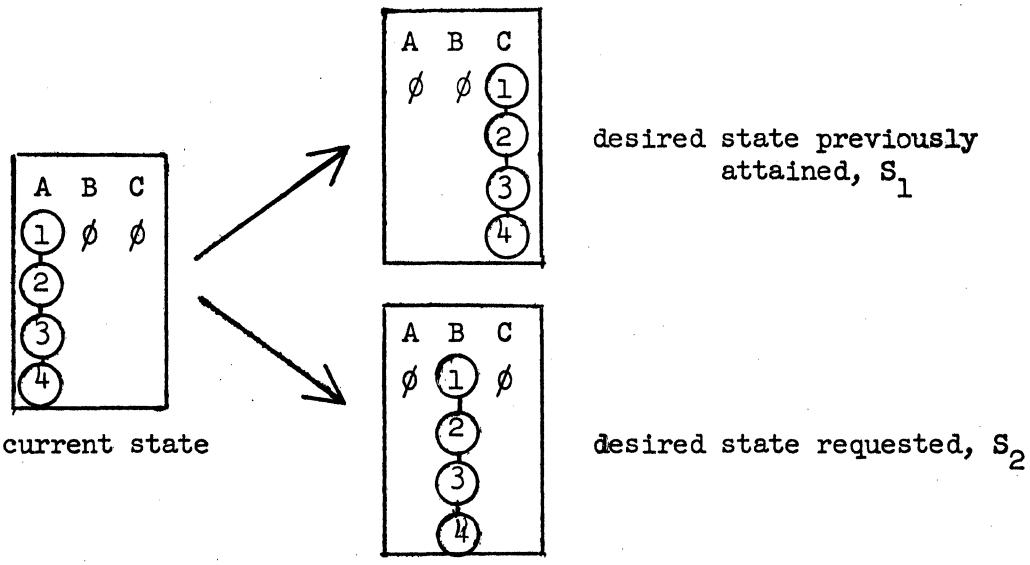
$$\begin{cases} \underline{1\ 2\ \ 1\ P_1}\ 1\ 2\ \ 1\ P_1\ 1 - - - - \\ \underline{B\ P_4\ C\ P_2\ A\ P_3}\ B\ P_4\ C - - - - \end{cases}$$

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

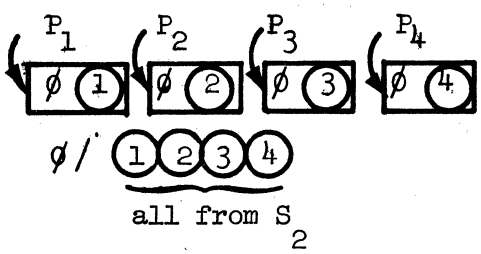
When the new programs are used to predict moves for the six-disk case, all turn out to be correct. In fact the parameterized program which has now been constructed will solve any n-disk case for three pegs, as long as the current state has n disks in column A and the desired state is disjunctive, i. e., either n disks in column B or in column C. 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 four-disk case is given with the desired state (in a non-disjunctive form) which has not been attained before, 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.

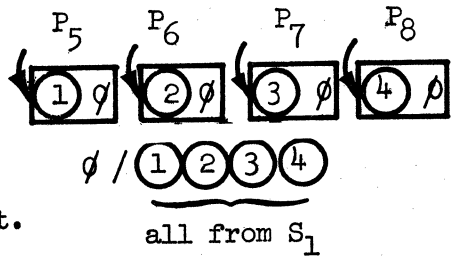
Comparison of the previously attained state and the newly requested state is given below.



Let us call the previously attained desired state S_1 and the newly requested desired state S_2. S_1 and S_2 are compared by the abstraction routine as follows: when elements A B C in S_1 are compared with elements A B C in S_2, S_1 and S_2 are found to be identical. Next, each element in S_1 which is itself a list is compared with the corresponding element in S_2. List A in S_1 and list A in S_2 are identical in content (both empty). For list B, the resulting abstractions are as follows: 0_1, 4_2



Similarly for list C: 4_1, 0_2:



∅ indicates no common element.

The task analyzer compares B and C by using the abstraction routine again, this time on the previously produced abstracts. Each of the three abstracts of B is paired with the corresponding abstact of C. The task analyzer then determines that the category of the change from S_1 to S_2 (or vice versa) is "interchange" of B and C. A similar analysis was made for the "exchange" routine, discussed under community unit operation.

Suppose, for simplicity, that such a program is already available in the conjecture generator so that the direction "interchange B and C" given by the task analyzer is understood. The conjecture generator then constructs two programs, this time without using its program-generating subunit; it copies the previously constructed conjecture programs (already parameterized) but modifies the second one by letting the "interchange" routine work on symbols B and C, i.e., whenever B appears, C is substituted and vice versa. When the programs are executed by the consequence generator, the resulting sequence of moves looks like this:

$$\underline{1\ 2\ 1\ P_1}\ \ 1\ 2\ 1\ P_1 - - - -$$

$$\underline{C\ P_5\ B\ P_6\ A\ P_7}\ \ C\ P_5 - - - -$$

When these suggested moves are output to the higher-level program and from there to the trainer, all of them turn out to be correct.[1]

Attaining the general rule

The principles involved in solving the puzzle can be stated by

[1] For the purpose of illustration, we neglected possible assistance from the planning mechanism discussed earlier.

considering columns A, B, and C to form a triangle, A, B, and C being its vertices. If the puzzle is given with a disjunctive desired state, transfer the smallest disk (disk (1)) on every other move, always moving it around the triangle in the same direction. On the remaining moves, make the only transfer possible that does not involve the smallest disk. However, if the puzzle is given in such a way that n disks must be transferred to a particular column, the first move is dictated by another rule. If the number of disks is odd, move first to the column to which the disks are to be moved, and if even, move first away from this column. The rest of the moves follow the preceding rule.

In our experiment, patterns generated by those programs which represent conjectures resemble the patterns described by the above statements, but some refinements are necessary in order to make them correspond exactly. The following might be given to the system at this point: The first cycle phase "1 2 1 P_5" can be transformed into "1 $\overline{1}$". The notation "$\overline{1}$" means anything other than (1). Then for the second sequence, "B P_4 C P_2 A P_3" or "C P_5 B P_6 A P_7", the system can be made to notice the correspondence between occurrences of 1 in the first sequence and the occurrences of "A", "B" and "C" in the second sequence. Similar correspondence can be established between $\overline{1}$ and parameters so that the parameter P can replace all the P_i's in the second sequence. The move $\boxed{\begin{matrix}\overline{1}\\ P\end{matrix}}$, which now occurs as every other move, is to mean any legal move which does not involve disk (1).

The rule when even and odd numbers of disks are involved will require a more sophisticated notion of evenness and oddness, a notion which will have to

be learned through experience or preprogrammed.

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 simplication. They observe scientists making deliberate over-simplifications 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 8-disk case of the Tower of Hanoi puzzle were given at the outset, it might have tried out the 2-disk or 3-disk puzzle of its own accord. Note that at this stage, solving the puzzle even by the exhaustive method is feasible. For the 3-disk puzzle, the exhaustive method would require 254 examinations of the current state configuration whereas such a method for the 8-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

Main theme of our research

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 in order 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 (3) points out, "In order for a program to be capable of learning something it must first be capable of being told it."

REFERENCES

(1) Kelly, J. L., Jr., and Selfridge, O. G. Sophistication in computers: a disagreement. IRE Transactions on Information Theory, 1962, IT-8, pp. 78-80.

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

(3) McCarthy, J. Programs with common sense. In Mechanisation of thought processes, Vol. 1. (National Physical Laboratory Symposium No. 10.) Her Majesty's Stationery Office, 1959, Pp. 75-84.

(4) Miller, G. A., Galanter, E., and Pribram, K. H. Plans and the structure of behavior. New York: Henry Holt and Co., 1960.

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

(6) Minsky, M. Steps toward artificial intelligence. Proceedings of the IRE, 1961, 49, pp. 8-30.

(7) Newell, A., (Ed.) Information processing language-V manual. Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1961.

(8) Newell, A., Shaw, J. C., and Simon, H. A. A general problem-solving program for a computer. Computers and Automation, 1959, 8, pp. 10-17.

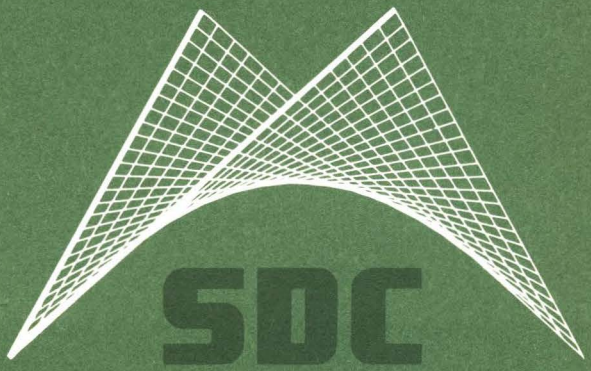
(9) Newell, A., Shaw, J. C., and Simon, H. A. A variety of intelligent learning in a general problem solver. In M. C. Yovits and S. Cameron (Eds.), Self-Organizing Systems, London, Pergamon Press, 1960. Pp. 153-189.

(10) Solomonoff, R. J. An inductive inference machine. IRE Convention Record, 1957, 5 (2), pp. 56-62.

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

(12) Tolman, E. C. Cognitive maps in rats and men. In his Behavior and psychological man, essays in motivation and learning. Berkeley and Los Angeles: Univ. of California Press, 1958. Pp. 241-264.

(13) Watanabe, S. Information-theoretical aspects of inductive and deductive inference. IBM Journal of Research and Development, 1960, 4, pp. 208-231.



SYSTEM DEVELOPMENT CORPORATION / 2500 COLORADO AVENUE / SANTA MONICA, CALIFORNIA