EMPIRICAL EXPLORATIONS WITH THE LOGIC THEORY MACHINE: A CASE STUDY IN HEURISTICS

by Allen Newell, J. C. Shaw, & H. A. Simon

This is a case study in problem-solving, representing part of a program of research on complex information-processing systems. We have specified a system for finding proofs of theorems in elementary symbolic logic, and by programming a computer to these specifications, have obtained empirical data on the problem-solving process in elementary logic. The program is called the Logic Theory Machine (LT); it was devised to learn how it is possible to solve difficult problems such as proving mathematical theorems, discovering scientific laws from data, playing chess, or understanding the meaning of English prose.

The research reported here is aimed at understanding the complex processes (heuristics) that are effective in problem-solving. Hence, we are not interested in methods that guarantee solutions, but which require vast amounts of computation. Rather, we wish to understand how a mathematician, for example, is able to prove a theorem even though he does not know when he starts how, or if, he is going to succeed.

This focuses on the pure theory of problem-research solving (Newell and Simon, 1956*a*). Previously we specified in detail a program for the Logic Theory Machine; and we shall repeat here only as much of that specification as is needed so that the reader can understand our data. In a companion study (Newell and Shaw, 1957) we consider how computers can be programmed to execute processes of the kinds called for by LT, a problem that is interesting in its own right. Similarly, we postpone to later papers a discussion of the implications of our work for the psychological theory of human thinking and problem-solving. Other areas of application

will readily occur to the reader, but here we will limit our attention to the nature of the problem-solving process itself.

Our research strategy in studying complex systems is to specify them in detail, program them for digital computers, and study their behavior empirically by running them with a number of variations and under a variety of conditions. This appears at present the only adequate means to obtain a thorough understanding of their behavior. Although the problem area with which the present system, LT, deals is fairly elementary, it provides a good example of a difficult problem—logic is a subject taught in college courses, and is difficult enough for most humans.

Our data come from a series of programs run on the JOHNNIAC, one of RAND's high-speed digital computers. We will describe the results of these runs, and analyze and interpret their implications for the problem-solving process.

The Logic Theory Machine in Operation

We shall first give a concrete picture of the Logic Theory Machine in operation. LT, of course, is a program, written for the JOHNNIAC, represented by marks on paper or holes in cards. However, we can think of LT as an actual physical machine and the operation of the program as the behavior of the machine. One can identify LT with JOHNNIAC after the latter has been loaded with the basic program, but before the input of data.

LT's task is to prove theorems in elementary symbolic logic, or more precisely, in the sentential calculus. The sentential calculus is a formalized system of mathematics, consisting of expressions built from combinations of basic symbols. Five of these expressions are taken as axioms, and there are rules of inference for generating new theorems from the axioms and from other theorems. In flavor and form elementary symbolic logic is much like abstract algebra. Normally the variables of the system are interpreted as sentences, and the axioms and rules of inference as formalizations of logical operations, *e.g.*, deduction. However, LT deals with the system as a purely formal mathematics, and we will have no further need of the interpretation. We need to introduce a smattering of the sentential calculus to understand LT's task.

There is postulated a set of *variables* $p, q, r, \ldots A, B, C, \ldots$, with which the sentential calculus deals. These variables can be combined into expressions by means of *connectives*. Given any variable p, we can form the expression "not-p." Given any two variables p and q, we can form the expression "p or q," or the expression "p implies q," where "or" and "implies" are the connectives. There are other connectives, for example "and," but we will not need them here. Once we have formed expressions,

these can be further combined into more complicated expressions. For example, we can form:[1]

"(p implies not-p) implies not-p." (2.01)

There is also given a set of expressions that are axioms. These are taken to be the universally true expressions from which theorems are to be derived by means of various rules of inference. For the sake of definiteness in our work with LT, we have employed the system of axioms, definitions, and rules that is used in the *Principia Mathematica,* which lists five axioms:

(p or p) implies p (1.2)
p implies (q or p) (1.3)
(p or q) implies (q or p) (1.4)
[p or (q or r)] implies [q or (p or r)] (1.5)
(p implies q) implies [(r or p) implies (r or q)]. (1.6)

Given some true theorems one can derive new theorems by means of three rules of inference: *substitution, replacement, and detachment.*

1. By the rule of substitution, any expression may be substituted for any variable in any theorem, provided the substitution is made throughout the theorem wherever that variable appears. For example, by substitution of "p or q" for "p," in the second axiom we get the new theorem:

(p or q) implies [q or (p or q)].

2. By the rule of replacement, a connective can be replaced by its definition, and *vice versa*, in any of its occurrences. By definition "p implies q" means the same as "not-p or q." Hence the former expression can always be replaced by the latter and *vice versa.* For example from axiom 1.3, by replacing "implies" with "or," we get the new theorem:

not-p or (q or p).

3. By the rule of detachment, if "A" and "A implies B" are theorems, then "B" is a theorem. For example, from:

(p or p) implies p,

and [(p or p) implies p] implies (p implies p),

we get the new theorem:

p implies p.

Given an expression to prove, one starts from the set of axioms and theorems already proved, and applies the various rules successively until

[1] For easy reference we have numbered axioms and theorems to correspond to their numbers in *Principia Mathematica,* 2nd ed., vol. 1, New York: by A. N. Whitehead and B. Russell, 1935.

the desired expression is produced. The proof is the sequence of expressions, each one validly derived from the previous ones, that leads from the axioms and known theorems to the desired expression.

This is all the background in symbolic logic needed to observe LT in operation. LT "understands" expressions in symbolic logic—that is, there is a simple code for punching expressions on cards so they can be fed into the machine. We give LT the five axioms, instructing it that these are theorems it can assume to be true. LT already knows the rules of inference and the definitions—how to substitute, replace, and detach. Next we give LT a single expression, say expression 2.01, and ask LT to find a proof for it. LT works for about 10 seconds and then prints out the following proof:

(p implies not-p) implies not-p	(theorem 2.01, to be proved)
1. (A or A) implies A	(axiom 1.2)
2. (not-A or not-A) implies not-A	(subs. of not-A for A)
3. (A implies not-A) implies not-A	(repl. of "or" with "implies")
4. (p implies not-p) implies not-p	(subs. of p for A; QED).

Next we ask LT to prove a fairly advanced theorem (Whitehead and Russell, 1935), theorem 2.45; allowing it to use all 38 theorems proved prior to 2.45. After about 12 minutes, LT produces the following proof:

not (p or q) implies not-p	(theorem 2.45, to be proved)
1. A implies (A or B)	(theorem 2.2)
2. p implies (p or q)	(subs. p for A, q for B in 1)
3. (A implies B) implies (not-B implies not-A)	(theorem 2.16)
4. [p implies (p or q)] implies [not (p or q) implies not-p]	[subs. p for A, (p or q) for B in 3]
5. not (p or q) implies not-p	(detach right side of 4, using 2; QED).

Finally, all the theorems prior to (2.31) are given to LT (a total of 28); and then LT is asked to prove:

$$[p \text{ or } (q \text{ or } r)] \text{ implies } [(p \text{ or } q) \text{ or } r]. \qquad (2.31)$$

LT works for about 23 minutes and then reports that it cannot prove (2.31), that it has exhausted its resources.

Now, what is there in this behavior of LT that needs to be explained? The specific examples given are difficult problems for most humans, and most humans do not know what processes they use to find proofs, if they find them. There is no known simple procedure that will produce such proofs. Various methods exist for verifying whether any given expression is

true or false; the best known procedure is the method of truth tables. But these procedures do not produce a proof in the meaning of Whitehead and Russell. One can invent "automatic" procedures for producing proofs. We will look at one briefly later, but these turn out to require computing times of the orders of thousands of years for the proof of (2.45).

We must clarify why such problems are difficult in the first place, and then show what features of LT account for its successes and failures. These questions will occupy the rest of this study.

Problems, Algorithms, and Heuristics

In describing LT, its environment, and its behavior we will make repeated use of three concepts. The first of these is the concept of *problem.* Abstractly, a person is given a problem if he is given a set of possible solutions, and a test for verifying whether a given element of this set is in fact a solution to his problem.

The reason why problems are problems is that the original set of possible solutions given to the problem-solver can be very large, the actual solutions can be dispersed very widely and rarely throughout it, and the cost of obtaining each new element and of testing it can be very expensive. Thus the problem-solver is not really "given" the set of possible solutions; instead he is given some process for generating the elements of that set in some order. This generator has properties of its own, not usually specified in stating the problem; *e.g.,* there is associated with it a certain cost per element produced, it may be possible to change the order in which it produces the elements, and so on. Likewise the verification test has costs and times associated with it. The problem can be solved if these costs are not too large in relation to the time and computing power available for solution.

One very special and valuable property that a generator of solutions sometimes has is a guarantee that if the problem has a solution, the generator will, sooner or later, produce it. We will call a process that has this property for some problem an *algorithm* for that problem. The guarantee provided by an algorithm is not an unmixed blessing, of course, since nothing has been specified about the cost or time required to produce the solutions. For example, a simple algorithm for opening a combination safe is to try all combinations, testing each one to see if it opens the safe. This algorithm is a typical problem-solving process: there is a generator that produces new combinations in some order, and there is a verifier that determines whether each new combination is in fact a solution to the problem. This search process is an algorithm because it is known that *some* combination will open the safe, and because the generator will exhaust all combinations in a finite interval of time. The algorithm is sufficiently expensive,

however, that a combination safe can be used to protect valuables even from people who know the algorithm.

A process that *may* solve a given problem, but offers no guarantees of doing so, is called a *heuristic*[2] for that problem. This lack of a guarantee is not an unmixed evil. The cost inflicted by the lack of guarantee depends on what the process costs and what algorithms are available as alternatives. For most run-of-the-mill problems we have only heuristics, but occasionally we have both algorithms and heuristics as alternatives for solving the same problem. Sometimes, as in the problem of finding maxima for simple differentiable functions, everyone uses the algorithm of setting the first derivative equal to zero; no one sets out to examine all the points on the line one by one as if it were possible. Sometimes, as in chess, everyone plays by heuristic, since no one is able to carry out the algorithm of examining all continuations of the game to termination.

The Problem of Proving Theorems in Logic

Finding a proof for a theorem in symbolic logic can be described as selecting an element from a generated set, as shown by Fig. 1. Consider the *set of all possible sequences of logic expressions*—call it E. Certain of these sequences, a very small minority, will be proofs. A proof sequence satisfies the following test:

Each expression in the sequence is either

1. One of the accepted theorems or axioms, or
2. Obtainable from one or two previous expressions in the sequence by application of one of the three rules of inference.

Call the *set of sequences that are proofs* P. Certain of the sequences in E have the *expression to be proved*—call it X, as their final expression. Call this set of sequences T_X. Then, to find a proof of a given theorem X means to select an element of E that belongs to the intersection of P and T_X. The set E is given implicitly by rules for generating new sequences of logic expressions.

The difficulty of proving theorems depends on the scarcity of elements in the intersection of P and T_X, relative to the number of elements in E. Hence, it depends on the cost and speed of the available generators that produce elements of E, and on the cost and speed of making tests that determine whether an element belongs to T_X or P. The difficulty also de-

[2] As a noun, "heuristic" is rare and generally means the art of discovery. The adjective "heuristic" is defined by Webster as: serving to discover or find out. It is in this sense that it is used in the phrase "heuristic process" or "heuristic method." For conciseness, we will use "heuristic" as a noun synonymous with "heuristic process." No other English word appears to have this meaning.

pends on whether generators can be found that guarantee that any element they produce automatically satisfies some of the conditions. Finally, as we shall see, the difficulty depends heavily on what heuristics can be found to guide the selection.

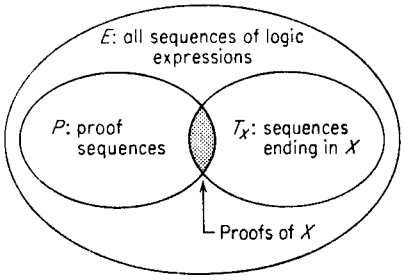

Figure 1. Relationships between E, P, and T_x.

A little reflection, and experience in trying to prove theorems, make it clear that proof sequences for specified theorems are rare indeed. To reveal more precisely why proving theorems is difficult, we will construct an algorithm for doing this. The algorithm will be based only on the tests and definitions given above, and not on any "deep" inferred properties of symbolic logic. Thus it will reflect the basic nature of theorem proving; that is, its nature prior to building up sophisticated proof techniques. We will call this algorithm the British Museum algorithm, in recognition of the supposed originators of procedures of this type.

The British Museum Algorithm

The algorithm constructs all possible proofs in a systematic manner, checking each time (1) to eliminate duplicates, and (2) to see if the final theorem in the proof coincides with the expression to be proved. With this algorithm the set of one-step proofs is identical with the set of axioms (*i.e.*, each axiom is a one-step proof of itself). The set of n-step proofs is obtained from the set of $(n-1)$-step proofs by making all the permissible substitutions and replacements in the expressions of the $(n-1)$-step proofs, and by making all the permissible detachments of pairs of expressions as permitted by the recursive definition of proof.[3]

Figure 2 shows how the set of n-step proofs increases with n at the very start of the proof-generating process. This enumeration only extends to replacements of "or" with "implies," "implies" with "or," and negation of variables (*e.g.*, "not-p" for "p"). No detachments and no complex substitutions (*e.g.*, "q or r" for "p") are included. No specializations have been made (*e.g.*, substitution of p for q in "p or q"). If we include the specializations, which take three more steps, the algorithm will generate

[3] A number of fussy but not fundamental points must be taken care of in constructing the algorithm. The phrase "all permissible substitutions" needs to be qualified, for there is an infinity of these. Care must be taken not to duplicate expressions that differ only in the names of their variables. We will not go into details here, but simply state that these difficulties can be removed. The essential feature in constructing the algorithm is to allow only one thing to happen in generating each new expression, i.e., one replacement, substitution of "not-p" for "p," etc.

an (estimated) additional 600 theorems, thus providing a set of proofs of 11 steps or less containing almost 1000 theorems, none of them duplicates.

In order to see how this algorithm would provide proofs of specified theorems, we can consider its performance on the sixty-odd theorems of chap. 2 of *Principia*. One theorem (2.01) is obtained in step (4) of the generation, hence is among the first 42 theorems proved. Three more (2.02, 2.03, and 2.04) are obtained in step (6), hence among the first 115. One more (2.05) is obtained in step (8), hence in the first 246. Only one more is included in the first 1000, theorem 2.07. The proofs of all the remainder require complex substitutions or detachment.

We have no way at present to estimate how many proofs must be generated to include proofs of all theorems of chap. 2 of *Principia*. Our best guess is that it might be a hundred million. Moreover, apart from the six theorems listed, there is no reason to suppose that the proofs of these theorems would occur early in the list.

Our information is too poor to estimate more than very roughly the times required to produce such proofs by the algorithm; but we can estimate times of about 16 minutes to do the first 250 theorems of Fig. 2 [*i.e.*, through step (8)] assuming processing times comparable with those in LT. The first part of the algorithm has an additional special property, which holds only to the point where detachment is first used; that no check for duplication is necessary. Thus the time of computing the first few thousand proofs only increases linearly with the number of theorems generated. For the theorems requiring detachments, duplication checks must be made, and the total computing time increases as the square of the number of expressions generated. At this rate it would take hundreds of thousands of years of computation to generate proofs for the theorems in chap. 2.

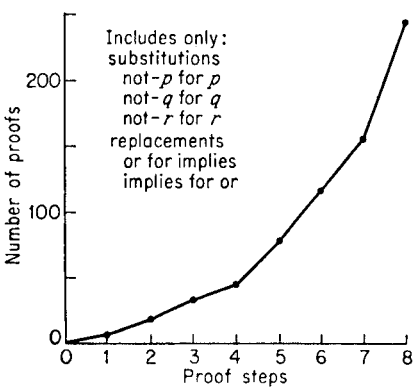

Figure 2. Number of proofs generated by first few steps of British Museum algorithm.

The nature of the problem of proving theorems is now reasonably clear. When sequences of expressions are produced by a simple and cheap (per element produced) generator, the chance that any particular sequence is the desired proof is exceedingly small. This is true even if the generator produces sequences that always satisfy the most complicated and restrictive of the solution conditions: that each is a proof of something. The set of sequences is so large, and the desired proof

so rare, that no practical amount of computation suffices to find proofs by means of such an algorithm.

The Logic Theory Machine

If LT is to prove any theorems at all it must employ some devices that alter radically the order in which possible proofs are generated, and the way in which they are tested. To accomplish this, LT gives up almost all the guarantees enjoyed by the British Museum algorithm. Its procedures guarantee neither that its proposed sequences are proofs of something, nor that LT will ever find the proof, no matter how much effort is spent. However, they *often* generate the desired proof in a reasonable computing time.

Methods

The major type of heuristic that LT uses we call a *method.* As yet we have no precise definition of a method that distinguishes it from all the other types of routines in LT. Roughly, a method is a reasonably self-contained operation that, if it works, makes a major and permanent contribution toward finding a proof. It is the largest unit of organization in LT, subordinated only to the executive routines necessary to coordinate and select the methods.

THE SUBSTITUTION METHOD

This method seeks a proof for the problem expression by finding an axiom or previously proved theorem that can be transformed, by a series of substitutions for variables and replacements of connectives, into the problem expression.

THE DETACHMENT METHOD

This method attempts, using the rule of detachment, to substitute for the problem expression a new subproblem which, if solved, will provide a proof for the problem expression. Thus, if the problem expression is B, the method of detachment searches for an axiom or theorem of the form "A implies B." If one is found, A is set up as a new subproblem. If A can be proved, then, since "A implies B" is a theorem, B will also be proved.

THE CHAINING METHODS

These methods use the transitivity of the relation of implication to create a new subproblem which, if solved, will provide a proof for the problem expression. Thus, if the problem expression is "a implies c," the method of forward chaining searches for an axiom or theorem of the form "a

implies *b*." If one is found, "*b* implies *c*" is set up as a new subproblem. Chaining backward works analogously: it seeks a theorem of the form "*b* implies *c*," and if one is found, "*a* implies *b*" is set up as a new subproblem.

Each of these methods is an independent unit. They are alternatives to one another, and can be used in sequence, one working on the subproblems generated by another. Each of them produces a major part of a proof. Substitution actually proves theorems, and the other three generate subproblems, which can become the intermediate expressions in a proof sequence.

These methods give no guarantee that they will work. There is no guarantee that a theorem can be found that can be used to carry out a proof by the substitution method, or a theorem that will produce a subproblem by any of the other three methods. Even if a subproblem is generated, there is no guarantee that it is part of the desired proof sequence, or even that it is part of any proof sequence (*e.g.*, it can be false). On the other hand, the generated methods do guarantee that any subproblem generated is part of a sequence of expressions that ends in the desired theorem (this is one of the conditions that a sequence be a proof). The methods also guarantee that each expression of the sequence is derived by the rules of inference from the preceding ones (a second condition of proof). What is not guaranteed is that the beginning of the sequence can be completed with axioms or previously proved theorems.

There is also no guarantee that the combination of the four methods, used in any fashion whatsoever and with unlimited computing effort, comprises a sufficient set of methods to prove all theorems. In fact, we have discovered a theorem [(2.13), "*p* or not-not-not-*p*"] which the four methods of LT cannot prove. All the subproblems generated for (2.13) after a certain point are false, and therefore cannot lead to a proof.

We have yet no general theory to explain why the methods transform LT into an effective problem-solver. That they do, in conjunction with the other mechanisms to be described shortly, will be demonstrated amply in the remainder of this study. Several factors may be involved. First, the methods organize the sequences of individual processing steps into larger units that can be handled as such. Each processing step can be oriented toward the special function it performs in the unit as a whole, and the units can be manipulated and organized as entities by the higher-level routines.

Apart from their "unitizing" effect, the methods that generate subproblems work "backward" from the desired theorem to axioms or known theorems rather than "forward" as did the British Museum algorithm. Since there is only one theorem to be proved, but a number of known true theorems, the efficacy of working backward may be analogous to the

ease with which a needle can find its way out of a haystack, compared with the difficulty of someone finding the lone needle in the haystack.

The Executive Routine

In LT the four methods are organized by an executive routine, whose flow diagram is shown in Fig. 3.

1. When a new problem is presented to LT, the substitution method is tried first, using all the axioms and theorems that LT has been told to assume, and that are now stored in a *theorem list.*

2. If substitution fails, the detachment method is tried, and as each new subproblem is created by a successful detachment, an attempt is made to prove the new subproblem by the substitution method. If substitution fails again, the subproblem is added to a *subproblem list.*

3. If detachment fails for all the theorems in the theorem list, the same cycle is repeated with forward chaining, and then with backward chaining: try to create a subproblem; try to prove it by the substitution method; if unsuccessful, put the new subproblem on the list. By the nature of the methods, if the substitution method ever succeeds with a single subproblem, the original theorem is proved.

4. If all the methods have been tried on the original problem and no proof has been produced, the executive routine selects the next untried subproblem from the subproblem list, and makes the same sequence of attempts with it. This process continues until (1) a proof is found, (2) the time allotted for finding a proof is used up, (3) there is no more available memory space in the machine, or (4) no untried problems remain on the subproblem list.

In the three examples cited earlier, the proof of (2.01) [(p implies not-p) implies not-p] was obtained by the substitution method directly, hence did not involve use of the subproblem list.

The proof of (2.45) [not (p or q) implies not-p] was achieved by an application of the detachment method followed by a substitution. This proof required LT to create a subproblem, and to use the substitution method on it. It did not require LT ever to select any sub-

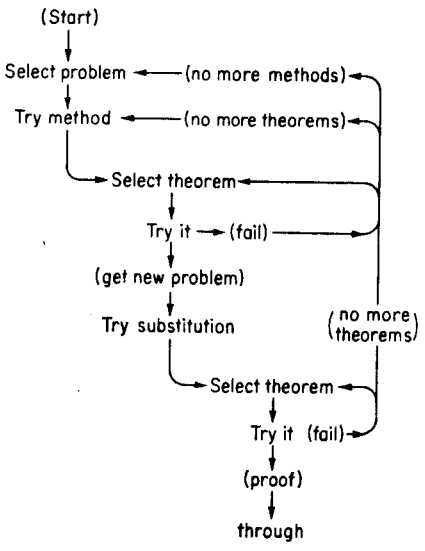

Figure 3. General flow diagram of LT.

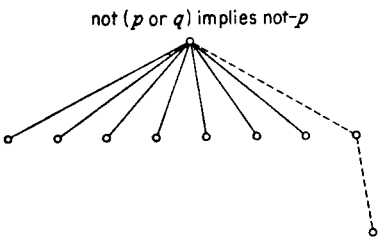

Figure 4. Subproblem tree of proof by LT of (2.45) (all previous theorems available).

problem from the subproblem list, since the substitution was successful. Figure 4 shows the *tree of subproblems* corresponding to the proof of (2.45). The subproblems are given in the form of a downward branching tree. Each node is a subproblem, the original problem being the single node at the top. The lines radiating down from a node lead to the new subproblems generated from the subproblem corresponding to the node. The proof sequence is given by the dashed line; the top link was constructed by the detachment method, and the bottom link by the substitution method. The other links extending down from the original problem lead to other subproblems generated by the detachment method (but not provable by direct substitution) prior to the time LT tried the theorem that leads to the final proof.

LT did not prove theorem 2.31, also mentioned earlier, and gave as its reason that it could think of nothing more to do. This means that LT had considered all subproblems on the subproblem list (there were six in this case) and had no new subproblems to work on. In none of the examples mentioned did LT terminate because of time or space limitations; however, this is the most common result in the cases where LT does not find a proof. Only rarely does LT run out of things to do.

This section has described the organization of LT in terms of methods. We have still to examine in detail why it is that this organization, in connection with the additional mechanisms to be described below, allows LT to prove theorems with a reasonable amount of computing effort.

The Matching Process

The times required to generate proofs for even the simplest theorems by the British Museum algorithm are larger than the times required by LT by factors ranging from five (for one particular theorem) to a hundred and upward. Let us consider an example from the earliest part of the generation, where we have detailed information about the algorithm. The 79th theorem generated by the algorithm (see Fig. 2) is theorem 2.02 of *Principia,* one of the theorems we asked LT to prove. This theorem, "p implies (q implies p)," is generated by the algorithm in about 158 seconds with a sequence of substitutions and replacements; it is proved by LT in about 10 seconds with the method of substitution. The reason for the difference becomes apparent if we focus attention on axiom 1.3, "p implies (q or p)," from which the theorem is derived in either scheme.

Figure 5 shows the tree of proofs of the first twelve theorems obtained from (1.3) by the algorithm. The theorem 2.02 is node (9) on the tree and is obtained by substitution of "not-q" for "q" in axiom 1.3 to reach node (5); and then by replacing the "(not-q or p)" by "(q implies p)" in (5) to get (9). The 9th theorem generated from axiom 1.3 is the 79th generated from the five axioms considered together.

This proof is obtained directly by LT using the following *matching* procedure. We compare the axiom with (9), the expression to be proved:

$$p \text{ implies } (q \text{ or } p) \qquad (1.3)$$
$$p \text{ implies } (q \text{ implies } p). \qquad (9)$$

First, by a direct comparison, LT determines that the main connectives are identical. Second, LT determines that the variables to the left of the main connectives are identical. Third, LT determines that the connectives within parentheses on the right-hand sides are different. It is necessary to replace the "or" with "implies," but in order to do this (in accordance with the definition of implies) there must be a negation sign before the variable that precedes the "or." Hence, LT first replaces the "q" on the right-hand side with "not-q" to get the required negation sign, obtaining (5). Now LT can change the "or" to "implies," and determines that the resulting expression is identical with (9).

The matching process allowed LT to proceed directly down the branch from (1) through (5) to (9) without even exploring the other branches. Quantitatively, it looked at only two expressions instead of eight, thus reducing the work of comparison by a factor of four. Actually, the saving is even greater, since the matching procedure does not deal with whole expressions, but with a single pair of elements at a time.

An important source of efficiency in the matching process is that it proceeds componentwise, obtaining at each step a feedback of the results of a substitution or replacement that can be used to guide the next step. This feedback keeps the search on the right branch of the tree of possible ex-

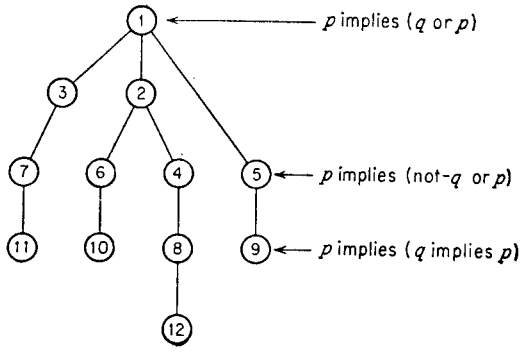

Figure 5. Proof tree of proof 2.02 by British Museum algorithm (using axiom 1.3).

pressions. It is not important for an efficient search that the goal be known from the beginning; it is crucial that hints of "warmer" or "colder" occur as the search proceeds.[4] Closely related to this feedback is the fact that where LT is called on to make a substitution or replacement at any step, it can determine immediately what variable or connective to substitute or replace by direct comparison with the problem expression, and without search.

Thus far we have assumed that LT knows at the beginning that (1.3) is the appropriate axiom to use. Without this information, it would begin matching with each axiom in turn, abandoning it for the next one if the matching should prove impossible. For example, if it tries to match the theorem against axiom 1.2, it determines almost immediately (on the second test) that "p or p" cannot be made into "p" by substitution. Thus, the matching process permits LT to abandon unprofitable lines of search as well as guiding it to correct substitutions and replacements.

MATCHING IN THE SUBSTITUTION METHODS

The matching process is an essential part of the substitution method. Without it, the substitution method is just that part of the British Museum algorithm that uses only replacements and substitutions. With it, LT is able, either directly or in combination with the other methods, to prove many theorems with reasonable effort.

To obtain data on its performance, LT was given the task of proving in sequence the first 52 theorems of *Principia*. In each case, LT was given the axioms plus all the theorems previously proved in chap. 2 as the material from which to work (regardless of whether LT had proved the theorems itself).[5]

Of the 52 theorems, proofs were found for a total 38 (73 per cent). These proofs were obtained by various combinations of methods, but the substitution method was an essential component of all of them. Seventeen of these proofs, almost a half, were accomplished by the substitution method alone. Subjectively evaluated, the theorems that were proved by

[4] The following analogy may be instructive. Changing the symbols in a logic expression until the "right" expression is obtained is like turning the dials on a safe until the right combination is obtained. Suppose two safes, each with ten dials and ten numbers on a dial. The first safe gives a signal (a "click") when any given dial is turned to the correct number; the second safe clicks only when all ten dials are correct. Trial-and-error search will open the first safe, on the average, in 50 trials; the second safe, in five billion trials.

[5] The version of LT used for seeking solutions of the 52 problems included a similarity test (see next section). Since the matching process is more important than the similarity test, we have presented the facts about matching first, using adjusted statistics. A notion of the sample sizes can be gained from Table 1. The sample was limited to the first 52 of the 67 theorems in chap. 2 of *Principia* because of memory limitations of JOHNNIAC.

the substitution method alone have the appearance of "corollaries" of the theorems they are derived from; they occur fairly close to them in the chapter, generally requiring three or fewer attempts at matching per theorem proved (54 attempts for 17 theorems).

The performance of the substitution method on the subproblems is somewhat different, due, we think, to the kind of selectivity implicit in the order of theorems in *Principia.* In 338 attempts at solving subproblems by substitution, there were 21 successes (6.2 per cent). Thus, there was about one chance in three of proving an original problem directly by the substitution method, but only about one chance in 16 of so proving a subproblem generated from the original problem.

MATCHING IN DETACHMENT AND CHAINING

So far the matching process has been considered only as a part of the substitution method, but it is also an essential component of the other three methods. In detachment, for example, a theorem of form "A implies B" is sought, where B is identical with the expression to be proved. The chances of finding such a theorem are negligible unless we allow some modification of B to make it match the theorem to be proved. Hence, once a theorem is selected from the theorem list, its right-hand subexpression is matched against the expression to be proved. An analogous procedure is used in the chaining methods.

We can evaluate the performance of the detachment and chaining methods with the same sample of problems used for evaluating the substitution method. However, a successful match with the former three methods generates a subproblem and does not directly prove the theorem. With the detachment method, an average of three new subproblems were generated for each application of the method; with forward chaining the average was 2.7; and with backward chaining the average was 2.2. For all the methods, this represents about one subproblem per 7½ theorems tested (the number of theorems available varied slightly).

As in the case of substitution, when these three methods were applied to the original problem, the chances of success were higher than when they were applied to subproblems. When applied to the original problem, the number of subproblems generated averaged eight to nine; when applied to subproblems derived from the original, the number of subproblems generated fell to an average of two or three.

In handling the first 52 problems in chap. 2 of *Principia,* 17 theorems were proved in one step—that is, in one application of substitution. Nineteen theorems were proved in two steps, 12 by detachment followed by substitution, and seven by chaining forward followed by substitution. Two others were proved in three steps. Hence, 38 theorems were proved in all. There are no two-step proofs by backward chaining, since, for two-step

proofs only, if there is a proof by backward chaining, there is also one by forward chaining. In 14 cases LT failed to find a proof. Most of these unsuccessful attempts were terminated by time or space limitations. One of these 14 theorems we know LT cannot prove, and one other we believe it cannot prove. Of the remaining twelve, most of them can be proved by LT if it has sufficient time and memory (see section on subproblems, however).

Similarity Tests and Descriptions

Matching eliminates enough of the trial and error in substitutions and replacements to make LT into a successful problem solver. Matching permeates all of the methods, and without it none of them would be useful within practical amounts of computing effort. However, a large amount of search is still used in finding the correct theorems with which matching works. Returning to the performance of LT in chap. 2, we find that the over-all chances of a particular match being successful are 0.3 per cent for substitution, 13.4 per cent for detachment, 13.8 per cent for forward chaining, and 9.4 per cent for backward chaining.

The amount of search through the theorem list can be reduced by interposing a screening process that will reject any theorem for matching that has low likelihood of success. LT has such a screening device, called the *similarity test.* Two logic expressions are defined to be similar if both their left-hand and right-hand sides are equal, with respect to, (1) the maximum number of *levels* from the main connective to any variable; (2) the number of *distinct* variables; and (3) the number of *variable places.* Speaking intuitively, two logic expressions are "similar" if they look alike, and look alike if they are similar. Consider for example:

$$(p \text{ or } q) \text{ implies } (q \text{ or } p) \qquad (1)$$
$$p \text{ implies } (q \text{ or } p) \qquad (2)$$
$$r \text{ implies } (m \text{ implies } r). \qquad (3)$$

By the definition of similarity, (2) and (3) are similar, but (1) is not similar to either (2) or (3).

In all of the methods LT applies the similarity tests to all expressions to be matched, and only applies the matching routine if the expressions are similar; otherwise it passes on to the next theorem in the theorem list. The similarity test reduces substantially the number of matchings attempted, as the numbers in Table 1 show, and correspondingly raises the probability of a match if the matching is attempted. The effect is particularly strong in substitution, where the similarity test reduces the matchings attempted by a factor of ten, and increases the probability of a successful match by a factor of ten. For the other methods attempted matchings were

TABLE 1 Statistics of Similarity Tests and Matching

Method	Theorems considered	Theorems similar	Theorems matched	Per cent similar of theorems considered	Per cent matched of theorems similar
Substitution	11,298	993	37	8.8	3.7
Detachment	1,591	406	210	25.5	51.7
Chain. forward	869	200	120	23.0	60.0
Chain. backward	673	146	63	21.7	43.2

reduced by a factor of four or five, and the probability of a match increased by the same factor.

These figures reveal a gross, but not necessarily a net, gain in performance through the use of the similarity test. There are two reasons why all the gross gain may not be realized. First, the similarity test is only a heuristic. It offers no guarantee that it will let through only expressions that will subsequently match. The similarity test also offers no guarantee that it will not reject expressions that would match if attempted. The similarity test does not often commit this type of error (corresponding to a type II statistical error), as will be shown later. However, even rare occurrences of such errors can be costly. One example occurs in the proof of theorem 2.07:

$$p \text{ implies } (p \text{ or } p). \qquad (2.07)$$

This theorem is proved simply by substituting p for q in axiom 1.3:

$$p \text{ implies } (q \text{ or } p). \qquad (1.3)$$

However, the similarity test, because it demands equality in the number of distinct variables on the right-hand side, calls (2.07) and (1.3) dissimilar because (2.07) contains only p while (1.3) contains p and q. LT discovers the proof through chaining forward, where it checks for a direct match before creating the new subproblem, but the proof is about five times as expensive as when the similarity test is omitted.

The second reason why the gross gain will not all be realized is that the similarity test is not costless, and in fact for those theorems which pass the test the cost of the similarity test must be paid in addition to the cost of the matching. We will examine these costs in the next section when we consider the effort LT expends.

Experiments have been carried out with a weaker similarity test, which compares only the number of variable places on both sides of the expression. This test will not commit the particular type II error cited above, and (2.07) is proved by substitution using it. Apart from this, the modifi-

cation had remarkably little effect on performance. On a sample of ten problems it admitted only 10 per cent more similar theorems and about 10 per cent more subproblems. The reason why the two tests do not differ more radically is that there is a high correlation among the descriptive measures.

Effort in LT

So far we have focused entirely on the performance characteristics of the heuristics in LT, except to point out the tremendous difference between the computing effort required by LT and by the British Museum algorithm. However, it is clear that each additional test, search, description, and the like, has its costs in computing effort as well as its gains in performance. The costs must always be balanced against the performance gains, since there are always alternative heuristics which could be added to the system in place of those being used. In this section we will analyze the computing effort used by LT. The memory space used by the various processes also constitutes a cost, but one that will not be discussed in this study.

MEASURING EFFORTS

LT is written in an interpretive language or pseudocode, which is described in the companion paper to this one. LT is defined in terms of a set of primitive operations, which, in turn, are defined by subroutines in JOHNNIAC machine language. These primitives provide a convenient unit of effort, and all effort measurements will be given in terms of total number of primitives executed. The relative frequencies of the different primitives are reasonably constant, and, therefore, the total number of primitives is an adequate index of effort. The average time per primitive is quite constant at about 30 milliseconds, although for very low totals (less than 1000 primitives) a figure of about 20 milliseconds seems better.

COMPUTING EFFORT AND PERFORMANCE

On *a priori* grounds we would expect the amount of computing effort required to solve a logic problem to be roughly proportional to the total number of theorems examined (*i.e.*, tested for similarity, if there is a similarity routine; or tested for matching, if there is not) by the various methods in the course of solving the problem. In fact, this turns out to be a reasonably good predictor of effort; but the fit to data is much improved if we assign greater weight to theorems considered for detachment and chaining than to theorems considered for substitution.

Actual and predicted efforts are compared below (with the full similarity test included, and excluding theorems proved by substitution) on the assumption that the number of primitives per theorem considered is twice as great for chaining as for substitution, and three times as great for de-

tachment. About 45 primitives are executed per theorem considered with the substitution method (hence 135 with detachment and 90 with chaining). As Table 2 shows, the estimates are generally accurate within a few per cent, except for theorem 2.06, for which the estimate is too low.

TABLE 2 Effort Statistics with "Precompute Description" Routine

Total primitives, thousands		
Theorem	Actual	Estimate
2.06	3.2	0.8
2.07	4.3	4.4
2.08	3.5	3.3
2.11	2.2	2.2
2.13	24.5	24.6
2.14	3.3	3.2
2.15	15.8	13.6
2.18	34.1	35.8
2.25	11.1	11.5

There is an additional source of variation not shown in the theorems selected for Table 2. The descriptions used in the similarity test must be computed from the logic expressions. Since the descriptions of the theorems are used over and over again, LT computes these at the start of a problem and stores the values with the theorems, so they do not have to be computed again. However, as the number of theorems increases, the space devoted to storing the precomputed descriptions becomes prohibitive, and LT switches to recomputing them each time it needs them. With recomputation, the problem effort is still roughly proportional to the total number of theorems considered, but now the number of primitives per theorem is around 70 for the substitution method, 210 for detachment, and 140 for chaining.

Our analysis of the effort statistics shows, then, that in the first approximation the effort required to prove a theorem is proportional to the number of theorems that have to be considered before a proof is found; the number of theorems considered is an effort measure for evaluating a heuristic. A good heuristic, by securing the consideration of the "right" theorems early in the proof, reduces the expected number of theorems to be considered before a proof is found.

EVALUATION OF THE SIMILARITY TEST

As we noted in the previous section, to evaluate an improved heuristic, account must be taken of any additional computation that the improvement introduces The net advantage may be less than the gross advantage,

or the extra computing effort may actually cancel out the gross gain in selectivity. We are now in a position to evaluate the similarity routines as preselectors of theorems for matching.

A number of theorems were run, first with the full similarity routine, then with the modified similarity routine (which tests only the number of variable places), and finally with no similarity test at all. We also made some comparisons with both precomputed and recomputed descriptions.

When descriptions are precomputed, the computing effort is less with the full similarity test than without it; the factor of saving ranged from 10 to 60 per cent (*e.g.*, 3534/5206 for theorem 2.08). However, if LT must recompute the descriptions every time, the full similarity test is actually more expensive than no similarity test at all (*e.g.*, 26,739/22,914 for theorem 2.45).

The modified similarity test fares somewhat better. For example, in proving (2.45) it requires only 18,035 primitives compared to the 22,914 for no similarity test (see the paragraph above). These comparisons involve recomputed descriptions; we have no figures for precomputed descriptions, but the additional saving appears small since there is much less to compute with the abridged than with the full test.

Thus the similarity test is rather marginal, and does not provide anything like the factors of improvement achieved by the matching process, although we have seen that the performance figures seem to indicate much more substantial gains. The reason for the discrepancy is not difficult to find. In a sense, the matching process consists of two parts. One is a testing part that locates the differences between elements and diagnoses the corrective action to be taken. The other part comprises the processes of substituting and replacing. The latter part is the major expense in a matching that works, but most of this effort is saved when the matching fails. Thus matching turns out to be inexpensive for precisely those expressions that the similarity test excludes.

Subproblems

LT can prove a great many theorems in symbolic logic. However, there are numerous theorems that LT cannot prove, and we may describe LT as having reached a plateau in its problem solving ability.

Figure 6 shows the amount of effort required for the problems LT solved out of the sample of 52. Almost all the proofs that LT found took less than 30,000 primitives of effort. Among the numerous attempts at proofs that went beyond this effort limit, only a few succeeded, and these required a total effort that was very much greater.

The predominance of short proofs is even more striking than the approximate upper limit of 30,000 primitives suggests. The proofs by substitution

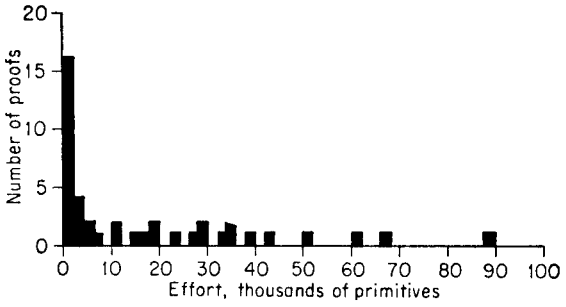

Figure 6. Distribution of LT's proofs by effort. Data include all proofs from attempts on the first 52 theorems in chap. 2 of *Principia*.

—almost half of the total—required about 1000 primitives or less each. The effort required for the longest proof—89,000 primitives—is some 250 times the effort required for the short proofs. We estimate that to prove the 12 additional theorems that we believe LT can prove requires the effort limit to be extended to about a million primitives.

From these data we infer that LT's power as a problem solver is largely restricted to problems of a certain class. While it is logically possible for LT to solve others by large expenditures of effort, major adjustments are needed in the program to extend LT's powers to essentially new classes of problems. We believe that this situation is typical: good heuristics produce differences in performance of large orders of magnitude, but invariably a "plateau" is reached that can be surpassed only with quite different heuristics. These new heuristics will again make differences of orders of magnitude. In this section we shall analyze LT's difficulties with those theorems it cannot prove, with a view to indicating the general type of heuristic that might extend its range of effectiveness.

The Subproblem Tree

Let us examine the proof of theorem 2.17 when all the preceding theorems are available. This is the proof that cost LT 89,000 primitives. It is reproduced below, using chaining as a rule of inference (each chaining could be expanded into two detachments, to conform strictly to the system of *Principia*).

(not-q implies not-p) implies (p-implies q)	(theorem 2.17, to be proved)
1. A implies not-not-A	(theorem 2.12)
2. p implies not-not-p	(subs. p for A in 1)
3. (A implies B) implies [(B implies C) implies (A implies C)]	(theorem 2.06)
4. (p implies not-not-p) implies [(not-not-p implies q) implies (p implies q)]	(subs. p for A, not-not-p for B, q for C in 3)

5. (not-not-p implies q) implies (p implies q)	(det. 4 from 3)
6. (not-A implies B) implies (not-B implies A)	(theorem 2.15)
7. (not-q implies not-p) implies (not-not-p implies q)	(subs. q for A, not-p for B)
8. (not-q implies not-p) implies (p implies q)	(chain 7 and 5; *QED*)

The proof is longer than either of the two given earlier. In terms of LT's methods it takes three steps instead of two or one: a forward chaining, a detachment, and a substitution. This leads to the not surprising notion, given human experience, that length of proof is an important variable in determining total effort: short proofs will be easy and long proofs difficult, and difficulty will increase more than proportionately with length of proof. Indeed, all the one-step proofs require 500 to 1500 primitives, while the number of primitives for two-step proofs ranges from 3000 to 50,000. Further, LT has obtained only six proofs longer than two steps, and these require from 10,000 to 90,000 primitives.

The significance of length of proof can be seen by comparing Fig. 7, which gives the proof tree for (2.17), with Fig. 4, which gives the proof tree for (2.45), a two-step proof. In going one step deeper in the case of (2.17), LT had to generate and examine many more subproblems. A comparison of the various statistics of the proofs confirms this statement: the problems are roughly similar in other respects (*e.g.*, in effort per theorem considered); hence the difference in total effort can be attributed largely to the difference in number of subproblems generated.

Let us examine some more evidence for this conclusion. Figure 8 shows the subproblem tree for the proof of (2.27) from the axioms, which is the only four-step proof LT has achieved to date. The tree reveals immediately

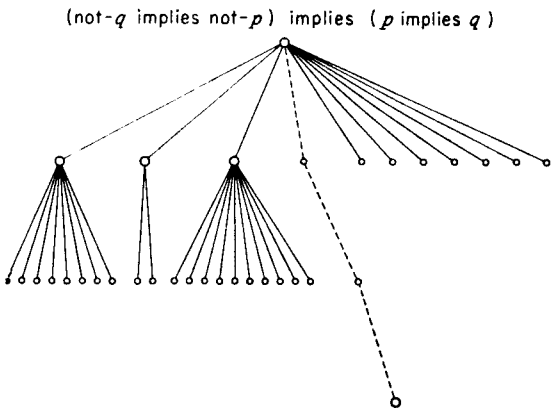

Figure 7. Subproblem tree of proof by LT of (2.17) (all previous theorems available).

why LT was able to find the proof. Instead of branching widely at each point, multiplying rapidly the number of subproblems to be looked at, LT in this case only generates a few subproblems at each point. It thus manages to penetrate to a depth of four steps with a reasonable amount of effort (38,367 primitives). If this tree had branched as the other two did, LT would have had to process about 250 subproblems before arriving at a proof, and the total effort would have been at least 250,000 primitives. The statistics quoted earlier on the effectiveness of subproblem generation support the general hypothesis that the number of subproblems to be examined increases more or less exponentially with the depth of the proof.

The difficulty is that LT uses an algorithmic procedure to govern its generation of subproblems. Apart from a few subproblems excluded by the type II errors of the similarity test, the procedure guarantees that all subproblems that can be generated by detachment and chaining will in fact be obtained (duplications are eliminated). LT also uses an algorithm to determine the order in which it will try to solve subproblems. The subproblems are considered in order of generation, so that a proof will not be missed through failure to consider a subproblem that has been generated.

Because of these systematic principles incorporated in the executive program, and because the methods, applied to a theorem list averaging 30 expressions in length, generate a large number of subproblems, LT must find a rare sequence that leads to a proof by searching through a very large set of such sequences. For proofs of one step, this is no problem at all; for proofs of two steps, the set to be examined is still of reasonable size in relation to the computing power available. For proofs of three steps, the size of the search already presses LT against its computing limits; and if one or two additional steps are added the amount of search required to

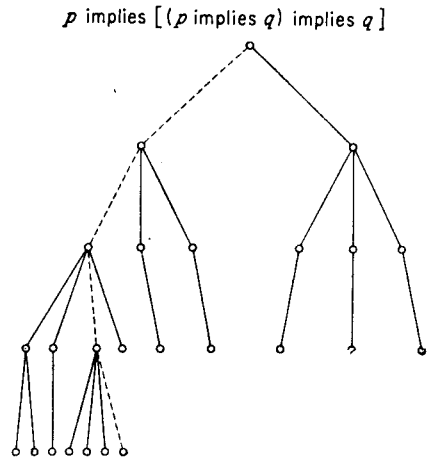

Figure 8. Subproblem tree of proof by LT of (2.27) (using the axioms).

find a proof exceeds any amount of computing power that could practically be made available.

The set of subproblems generated by the Logic Theory Machine, however large it may seem, is exceedingly selective and rich in proofs compared with the set through which the British Museum algorithm searches. Hence, the latter algorithm could find proofs in a reasonable time for only the simplest theorems, while proofs for a much larger number are accessible with LT. The line dividing the possible from the impossible for any given problem-solving procedure is relatively sharp; hence a further increase in problem-solving power, comparable to that obtained in passing from the British Museum algorithm to LT, will require a corresponding enrichment of the heuristic.

Modification of the Logic Theory Machine

There are many possible ways to modify LT so that it can find proofs of more than two steps in a way which has reason and insight, instead of by brute force. First, the unit cost of processing subproblems can be substantially reduced so that a given computing effort will handle many more subproblems. (This does not, perhaps, change the "brute force" character of the process, but makes it feasible in terms of effort.) Second, LT can be modified so that it will select for processing only subproblems that have a high probability of leading to a proof. One way to do this is to screen subproblems before they are put on the subproblem list, and eliminate the unlikely ones altogether. Another way is to reduce selectively the number of subproblems generated.

For example, to reduce the number of subproblems generated, we may limit the lists of theorems available for generating them. That this approach may be effective is suggested by the statistics we have already cited, which show that the number of subproblems generated by a method per theorem examined is relatively constant (about one subproblem per seven theorems).

An impression of how the number of available theorems affects the generation of subproblems may be gained by comparing the proof trees of (2.17) (Fig. 7) and (2.27) (Fig. 8). The broad tree for (2.17) was produced with a list of twenty theorems, while the deep tree for (2.27) was produced with a list of only five theorems. The smaller theorem list in the latter case generated fewer subproblems at each application of one of the methods.

Another example of the same point is provided by two proofs of theorem 2.48 obtained with different lists of available theorems. In the one case, (2.48) was proved starting with all prior theorems on the theorem list; in the other case it was proved starting only with the axioms and theorem 2.16. We had conjectured that the proof would be more

difficult to obtain under the latter conditions, since a longer proof chain would have to be constructed than under the former. In this we were wrong: with the longer theorem list, LT proved theorem 2.48 in two steps, employing 51,450 primitives of effort. With the shorter list, LT proved the theorem in three steps, but with only 18,558 primitives, one-third as many as before. Examination of the first proof shows that the many "irrelevant" theorems on the list took a great deal of processing effort. The comparison provides a dramatic demonstration of the fact that a problem solver may be encumbered by too much information, just as he may be handicapped by too little.

We have only touched on the possibilities for modifying LT, and have seen some hints in LT's current behavior about their potential effectiveness. All of the avenues mentioned earlier appear to offer worthwhile modifications of the program. We hope to report on these explorations at a later time.

Conclusion

We have provided data on the performance of a complex information processing system that is capable of finding proofs for theorems in elementary symbolic logic. We have used these data to analyze and illustrate the difference between systematic, algorithmic processes, on the one hand, and heuristic, problem-solving processes, on the other. We have shown how heuristics give the program power to solve problems in a reasonable computing time that could be solved algorithmically only in large numbers of years. Finally, we have assessed the limitations of the present program of the Logic Theory Machine and have indicated some of the directions that improvement would have to take to extend its powers to problems at new levels of difficulty.

Our explorations of the Logic Theory Machine represent a step in a program of research on complex information processing systems that is aimed at developing a theory of such systems and applying that theory to such fields as computer programming and human learning and problem-solving.