part 1
Artificial Intelligence

The purpose of this volume is to inform the nonspecialist about current research on intelligent behavior by computer—not by exhortation or reinterpretation, but by the collection of key scientific research reports which collectively represent the state of progress in this field. Each is an important paper for an informed understanding of research in artificial intelligence. In this introduction we hope to provide the reader with a set of guidelines to a thoughtful reading of the collected papers.

What Is a Computer? Is It Just a "Number Factory"?

In the popular conception, a computer is a high-speed number calculator. This view is only partly correct. A digital computer is, in fact, a general symbol-processing device, capable of performing any well-defined process for the manipulation and transformation of information.

All general-purpose digital computers are basically alike. They have:

1. One or more "input" devices for transforming symbolic information external to the machine into internally usable form. These internal forms are the *symbols* which the machine manipulates. A punched-card reader is an example of an input device.
2. One or more "output" devices for transforming the internal symbols back into external form. The computer's printer is an example of an output device.

3. One or more "memory" devices capable of storing symbols before, during, and after processing.

4. An "arithmetic unit." One of the possible interpretations that can be given to a computer's symbols is the interpretation as *numbers*. The arithmetic unit is a piece of electronic gear which will operate upon these numbers to produce (under the numerical interpretation) sums, differences, products, etc. Most of the computation described in this volume is nonnumeric. For example, the chess pieces manipulated by the Newell-Shaw-Simon Chess Player are represented and handled as symbols, not numbers.

5. A "control unit." The control unit is the executive of the computer organization. It is wired to understand and obey a repertory of *instructions* (or commands), calling the other units into action when necessary. The instructions are generally elementary processes, *e.g.*, fetch a symbol from a specified place in memory, return a symbol to some place in memory, shift a symbol a certain number of places to the left or right in "working storage."

A very important instruction, "compare and transfer control," enables the computer to make a simple two-choice decision—to take one of two specified courses of action depending on the information found in some cell of the memory. By cascading these simple decisions, highly complex decisions can be fashioned.

Information processes more complicated than those "wired into" the computer can be carried out by means of a sequence of the elementary instructions, called a *program*. The program is the precise statement of the information process that the user desires the machine to carry out. A computer's program is stored in the memory along with all the other problem information and data. One part of a program can call in another part of the program from the memory to the working storage and alter it. The general-purpose digital computer can do any information processing task for which a program can be written. The same computer which one moment is computing a company's payroll may in the next moment be computing aircraft designs or insurance premiums. Any program for a general-purpose computer effectively converts this general-purpose machine into a special-purpose machine for doing that task intended by the user who wrote the program.

Is It Possible for Computing Machines to Think?

No—if one defines thinking as an activity peculiarly and exclusively *human*. Any such behavior in machines, therefore, would have to be called thinking-like behavior.

No—if one postulates that there is something in the essence of thinking which is *inscrutable, mysterious, mystical.*

Yes—if one admits that the question is to be answered by *experiment and observation,* comparing the behavior of the computer with that behavior of human beings to which the term "thinking" is generally applied.

We regard the two negative views as unscientifically dogmatic. The positive, or empirical, view is explored with cogency by Turing in an article reprinted in this volume. Armer, in another reprinted article, qualifies the positive view by pointing out that there exists a continuum of intelligent behavior, that the question of how far we can push machines out along that continuum is to be answered by research, not dogma. We might add one further qualification: to assert that thinking machines are possible is not necessarily to assert that thinking machines with human capabilities already exist (or that they will exist in the near future). The reader of this volume is invited to form a judgment on the matter. The reports reprinted here constitute, we think, the best evidence available on the subject at present.

What, then, is the goal of artificial intelligence research? As we interpret the field, it is this: *to construct computer programs which exhibit behavior that we call "intelligent behavior" when we observe it in human beings.*

Because this research area is still in the formative stage of its development, many different research paths are being explored. Our goal definition may be too ambitious for some researchers, not ambitious enough for others (chiefly because it is tied to human behavior).

Many of the research projects reported in Part 1 achieve this goal within their special problem areas. Shall we call this computer behavior "thinking," or shall we not? Perhaps this is an individual's choice. In our opinion, it is neither an important nor a fruitful topic for debate.

But Doesn't a Computer Do Exactly What It Is Told To Do and No More?

Commenting on this familiar question, a well-known researcher in the field had this to say:

This statement—that computers can do only what they are programmed to do—is intuitively obvious, indubitably true, and supports none of the implications that are commonly drawn from it.

A human being can think, learn, and create because the program his biological endowment gives him, together with the changes in *that program produced by interaction with his environment after birth, enables him to think, learn, and create. If a computer thinks, learns, and creates, it will be by virtue of a program that endows it with these capacities. Clearly this will not be a program—any more than the human's is—that calls for highly stereotyped and repetitive behavior independent of the stimuli coming from the environment and the task to be completed. It will be a program that makes the system's behavior highly conditional on the task environment—on the task goals and on the clues extracted from the environment that indicate whether progress is being made toward those goals. It will be a program that analyzes, by some means, its own performance, diagnoses its failures, and makes changes that enhance its future effectiveness* (Simon, 1960, p. 25).

Similarly, it is wrong to conclude that a computer can exhibit behavior no more intelligent than its human programmer and that this astute gentleman can accurately predict the behavior of his program. These conclusions ignore the enormous complexity of information processing possible in problem-solving and learning machines. They presume that, because the programmer can write down (as programs) general prescriptions for adaptive behavior in such mechanisms, he can comprehend the remote consequences of these mechanisms after the execution of millions of information processing operations and the interaction of these mechanisms with a task environment. And, more importantly, they presume that he can perform the same complex information processing operations equally well with the device within his skull.

Is It True That a Computer Will Be a Chess Champion Because the Computer Is So Fast That It Can Examine All Possible Moves and Their Consequences?

This view of the problem-solving potential of computers rests on the assumption that, because computers are so fast, they can "think of everything." This kind of computing might be called *brute-force* computing. Brute-force programs generally have a simple structure, employing exhaustive enumeration of possibilities and exhaustive search. Is brute-force computing a general method for handling problems that are usually thought of as having some "intellectual content"?

To answer this question, we must look first at what *a problem* is. A problem exists for a problem-solver when he is faced with the task of choosing one of a set of alternatives placed before him by the problem environment. The problem-solver has no problem if the environment presents him with only one alternative; he must take that alternative. What is troublesome about alternatives is not so much their number as their consequences. Alternatives usually have elaborate consequences, which need to be evaluated before one alternative is chosen. The formal expression of this notion leads us to the so-called *maze model of a problem.* Let us look at this in an example.

Consider the problem of choosing a move at some point in a game of chess. If the position allows the player only one alternative, there is no problem—the move is *forced.* If, however, there is a genuine problem, the decision can be made by examining the immediate and remote consequences of selecting particular alternatives—the moves opened up to the opponent, the possible replies to these moves, etc. This "tree of possibilities" is pictured in Fig. 1.

In principle, this tree can be completely elaborated; the end points can be identified as wins, losses, or draws; and a strategy can be employed to determine the best alternative available at the top of the tree.

Since this procedure can in principle be programmed on modern, high-speed computers, why is chess still an interesting game? Why are computers not unbeatable champions at chess?

The answer is simple: the size of the chess maze is enormous. It has been estimated that there are about 10^{120} different paths through a complete chess maze (give or take, perhaps, many powers of ten). Even under the most generous assumptions about the power of modern computing machinery, now or in the future, it is beyond the limits of plausibility that a computer will ever be able to play "optimum" chess by the exhaustive strategy mentioned above.

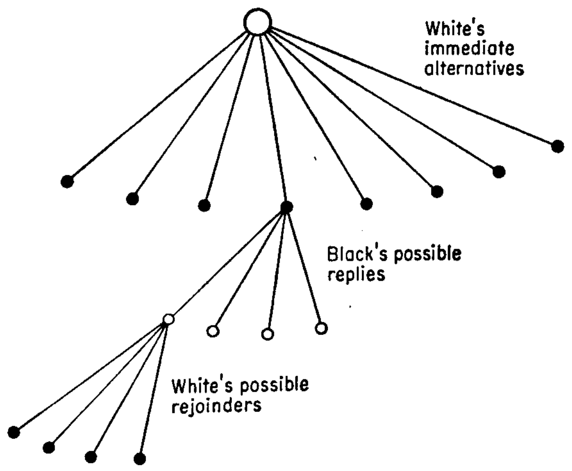


Figure 1.

Brute-force computing through problem mazes (for any but the most trivial problems) just won't do. Problem-solving by this method is beyond the realm of practical possibility.

How, then, are we to construct an intelligent problem-solver?

It appears that the clue to intelligent behavior, whether of men or machines, is *highly selective search,* the drastic pruning of the tree of possibilities explored. *For a computer program to behave intelligently, it must search problem mazes in a highly selective way, exploring paths relatively fertile with solutions and ignoring paths relatively sterile.*

What Is a Heuristic Program?

A *heuristic* (*heuristic rule, heuristic method*) is a rule of thumb, strategy, trick, simplification, or any other kind of device which drastically limits search for solutions in large problem spaces. Heuristics do not guarantee optimal solutions; in fact, they do not guarantee any solution at all; *all that can be said for a useful heuristic is that it offers solutions which are good enough most of the time.* A *heuristic program* is a program that employs heuristics in solving complex problems.

Heuristic methods have sometimes been contrasted with *algorithmic methods* for finding problem solutions, and a certain amount of intellectual blood has been shed unnecessarily on this battlefield. Without getting into the subtleties of the disagreement, we observe that the term "algorithm" is used with considerable ambiguity in mathematics and logic. Under one commonly held definition, algorithms are decision procedures which are *guaranteed* to produce the solution being sought, given enough time. The brute-force program described above for playing chess is such an algorithm. But algorithms (under this concept) are known, or practical, for only a very small subset of all interesting problems one would like to have computers solve. Over the spectrum of the broader class, heuristic methods seem to offer more general applicability.

The payoff in using heuristics is greatly reduced search and, therefore, practicality. Often, but not always, a price is paid: by drastic search limitations, sometimes the best solution (indeed, *any or all* solutions) may be overlooked.

Heuristics come in at least two varieties: special-purpose and general-purpose. Let us examine these by example:

1. The chess duffer might typically use this rule of thumb: Stop exploring any sequence that puts the queen in immediate danger of being captured. This is a special-purpose chess heuristic. It is useful

to the duffer because it keeps him out of one kind of trouble. By using this especially crude search-limiting device, the duffer will never discover those exciting queen-sacrifice combinations which get "!!" annotations in books on chess.

2. In proving theorems, a mathematician usually works backward from the theorem he is trying to prove to known theorems or axioms, instead of working forward from known expressions, using the rules of inference, until he stumbles on the theorem he has to prove. Under certain conditions, "working backward" is a powerful general heuristic for utilizing information in the problem to guide search for the solution.

3. A useful rule of thumb used by human beings in most of their problem-solving is this: Attack a new problem by methods that have solved similar problems in the past. The criteria for "similarity" may themselves be heuristic. If the environment is in a kind of steady state with respect to problem types, this heuristic may be very useful. In environments demanding a high degree of innovative problem-solving, this heuristic will hinder rather than facilitate problem-solving.

4. Two general-purpose heuristic problem-solving methods commonly employed in human reasoning are *means-ends analysis* and *planning*. In means-ends analysis, an initial problem state is transformed into a target state by selecting and applying operations which, step by step, reduce the difference between the states. In the planning method, a simplified statement of the original problem is constructed, and means-ends analysis is applied to this new, simpler problem. The result is a set of plans (guesses at possible operator sequences), hopefully one of which will work, *i.e.*, solve the original problem. Means-ends analysis is discussed in detail in the reprinted article on the General Problem Solver (GPS).

What Are Some Unsolved Problems of Artificial Intelligence Research?

In an area so new and exploratory, most of the problems remain unsolved, indeed unattacked. At this stage, it is not easy even to identify and state the problems, except in a very general way. We offer some examples of problems we think are ripe for attack:

1. The learning of heuristics. A puzzling, fascinating, and extremely important question is this: How can computers (and how do people) learn new heuristic methods and rules, both special-purpose and general-purpose? At the moment, our knowledge of learning mechanisms for problem-solving programs is rudimentary. Any significant breakthrough in this area would offer the promise of enabling

us to "bootstrap" our way into very much more powerful problem-solving programs.

2. Inductive inference. Artificial intelligence currently is strong on deductive inference, weak on inductive inference. Yet, in the melting pot of everyday intelligence, induction is certainly the more significant ingredient. One way of looking at the problem is that we need programs which will in some sense induce internally stored "models" of external environments—models from which the programs can make valid and useful predictions of future environmental states. Looked at in another way, this is the problem of hypothesis formation by machine. It is the general pattern recognition problem. Today we know very little about this crucial problem.

3. Understanding natural language. A problem of great theoretical and practical interest is that of constructing a program to understand communication in natural language (the word "understand" is here used with its full human connotation). To put it simply, one would like to be able to engage in a dialog with a computer—a dialog in which the computer will hold up its end of the conversation adaptively, intelligently, with understanding. Research on question-answering programs (*e.g.*, the BASEBALL program reprinted in this volume) is a good start. There is much that can be transferred from research on mechanical translation, on information retrieval, on models of human associative memory, and other areas of information science. The problem is ripe for intensive, interdisciplinary study.

What Are the Limits of Artificial Intelligence Research?

No one can answer this question today.

Perhaps the question has more fascination than importance. In terms of the continuum of intelligence suggested by Armer, the computer programs we have been able to construct are still at the low end. What is important is that we continue to strike out in the direction of the milestone that represents the capabilities of human intelligence. Is there any reason to suppose that we shall never get there? None whatever. Not a single piece of evidence, no logical argument, no proof or theorem has ever been advanced which demonstrates an insurmountable hurdle along the continuum.

Today, despite our ignorance, we can point to that biological milestone, the thinking brain, in the same spirit as the scientists many hundreds of years ago pointed to the bird as a demonstration in nature that mechanisms heavier than air could fly.

section 1

Can a Machine Think?

One of the unfortunate circumstances of the early post-World War II history of computer technology was that the high-speed electronic digital computer became styled as a "giant brain." The widespread use of this term by science popularizers, science-fiction writers, and their advertising-agency counterparts caused a vigorous and widespread reaction among the members of the rapidly expanding computer profession. The experience of the time was that it was difficult to program computers to do even the simplest data processing and computational jobs (*e.g.*, computerizing a payroll procedure).

To the theorists, however, the question of thinking by machine still held considerable interest. They were interested in defining the question more clearly and in discerning the existence or nonexistence of various kinds of theoretical upper bounds on the intelligence of computing devices. One of the best-known papers to emerge from these deliberations was by the famous English mathematician and logician, A. M. Turing, reprinted in this section. Turing's paper appeared five years before concrete developments in intelligent behavior by machine began to occur. Yet it remains today one of the most cogent and thorough discussions in the literature on the general question "Can a machine think?"

Turing takes a behavioristic posture relative to the question. The question is to be decided by an unprejudiced comparison of the alleged "thinking behavior" of the machine with normal "thinking behavior" in human beings. He proposes an experiment—commonly called "Turing's test"—in which the unprejudiced comparison could be made. Though the test has flaws, it is the best that has been proposed to date.

A. M. Turing died suddenly in 1954 after a short but brilliant career. Shortly before his death, while at the National Physical Laboratory in England, he completed the design of one of the world's first modern high-speed digital computers.

COMPUTING MACHINERY AND INTELLIGENCE

by A. M. Turing

1. The Imitation Game

I propose to consider the question, "Can machines think?" This should begin with definitions of the meaning of the terms "machine" and "think." The definitions might be framed so as to reflect so far as possible the normal use of the words, but this attitude is dangerous. If the meaning of the words "machine" and "think" are to be found by examining how they are commonly used it is difficult to escape the conclusion that the meaning and the answer to the question, "Can machines think?" is to be sought in a statistical survey such as a Gallup poll. But this is absurd. Instead of attempting such a definition I shall replace the question by another, which is closely related to it and is expressed in relatively unambiguous words.

The new form of the problem can be described in terms of a game which we call the "imitation game." It is played with three people, a man (A), a woman (B), and an interrogator (C) who may be of either sex. The interrogator stays in a room apart from the other two. The object of the game for the interrogator is to determine which of the other two is the man and which is the woman. He knows them by labels X and Y, and at the end of the game he says either "X is A and Y is B" or "X is B and Y is A." The interrogator is allowed to put questions to A and B thus:

C: Will X please tell me the length of his or her hair?
Now suppose X is actually A, then A must answer. It is A's object in the game to try and cause C to make the wrong identification. His answer might therefore be:

"My hair is shingled, and the longest strands are about nine inches long."

In order that tones of voice may not help the interrogator the answers should be written, or better still, typewritten. The ideal arrangement is to have a teleprinter communicating between the two rooms. Alternatively the question and answers can be repeated by an intermediary. The object of the game for the third player (B) is to help the interrogator. The best strategy for her is probably to give truthful answers. She can add such things as "I am the woman, don't listen to him!" to her answers, but it will avail nothing as the man can make similar remarks.

We now ask the question, "What will happen when a machine takes the part of A in this game?" Will the interrogator decide wrongly as often when the game is played like this as he does when the game is played between a man and a woman? These questions replace our original, "Can machines think?"

2. *Critique of the New Problem*

As well as asking, "What is the answer to this new form of the question," one may ask, "Is this new question a worthy one to investigate?" This latter question we investigate without further ado, thereby cutting short an infinite regress.

The new problem has the advantage of drawing a fairly sharp line between the physical and the intellectual capacities of a man. No engineer or chemist claims to be able to produce a material which is indistinguishable from the human skin. It is possible that at some time this might be done, but even supposing this invention available we should feel there was little point in trying to make a "thinking machine" more human by dressing it up in such artificial flesh. The form in which we have set the problem reflects this fact in the condition which prevents the interrogator from seeing or touching the other competitors, or hearing their voices. Some other advantages of the proposed criterion may be shown up by specimen questions and answers. Thus:

Q: Please write me a sonnet on the subject of the Forth Bridge.
A: Count me out on this one. I never could write poetry.
Q: Add 34957 to 70764.
A: (Pause about 30 seconds and then give as answer) 105621.
Q: Do you play chess?
A: Yes.
Q: I have K at my K1, and no other pieces. You have only K at K6 and R at R1. It is your move. What do you play?
A: (After a pause of 15 seconds) R-R8 mate.

The question and answer method seems to be suitable for introducing almost any one of the fields of human endeavour that we wish to include.

We do not wish to penalise the machine for its inability to shine in beauty competitions, nor to penalise a man for losing in a race against an aeroplane. The conditions of our game make these disabilities irrelevant. The "witnesses" can brag, if they consider it advisable, as much as they please about their charms, strength or heroism, but the interrogator cannot demand practical demonstrations.

The game may perhaps be criticised on the ground that the odds are weighted too heavily against the machine. If the man were to try and pretend to be the machine he would clearly make a very poor showing. He would be given away at once by slowness and inaccuracy in arithmetic. May not machines carry out something which ought to be described as thinking but which is very different from what a man does? This objection is a very strong one, but at least we can say that if, nevertheless, a machine can be constructed to play the imitation game satisfactorily, we need not be troubled by this objection.

It might be urged that when playing the "imitation game" the best strategy for the machine may possibly be something other than imitation of the behaviour of a man. This may be, but I think it is unlikely that there is any great effect of this kind. In any case there is no intention to investigate here the theory of the game, and it will be assumed that the best strategy is to try to provide answers that would naturally be given by a man.

3. *The Machines Concerned in the Game*

The question which we put in §1 will not be quite definite until we have specified what we mean by the word "machine." It is natural that we should wish to permit every kind of engineering technique to be used in our machines. We also wish to allow the possibility than an engineer or team of engineers may construct a machine which works, but whose manner of operation cannot be satisfactorily described by its constructors because they have applied a method which is largely experimental. Finally, we wish to exclude from the machines men born in the usual manner. It is difficult to frame the definitions so as to satisfy these three conditions. One might for instance insist that the team of engineers should be all of one sex, but this would not really be satisfactory, for it is probably possible to rear a complete individual from a single cell of the skin (say) of a man. To do so would be a feat of biological technique deserving of the very highest praise, but we would not be inclined to regard it as a case of "constructing a thinking machine." This prompts us to abandon the requirement that every kind of technique should be permitted. We are the more ready to do so in view of the fact that the present interest in "thinking machines" has been aroused by a particular kind of machine, usually called an "electronic computer" or "digital computer." Following this suggestion we only permit digital computers to take part in our game.

This restriction appears at first sight to be a very drastic one. I shall attempt to show that it is not so in reality. To do this necessitates a short account of the nature and properties of these computers.

It may also be said that this identification of machines with digital computers, like our criterion for "thinking," will only be unsatisfactory if (contrary to my belief), it turns out that digital computers are unable to give a good showing in the game.

There are already a number of digital computers in working order, and it may be asked, "Why not try the experiment straight away? It would be easy to satisfy the conditions of the game. A number of interrogators could be used, and statistics compiled to show how often the right identification was given." The short answer is that we are not asking whether all digital computers would do well in the game nor whether the computers at present available would do well, but whether there are imaginable computers which would do well. But this is only the short answer. We shall see this question in a different light later.

4. Digital Computers

The idea behind digital computers may be explained by saying that these machines are intended to carry out any operations which could be done by a human computer. The human computer is supposed to be following fixed rules; he has no authority to deviate from them in any detail. We may suppose that these rules are supplied in a book, which is altered whenever he is put on to a new job. He has also an unlimited supply of paper on which he does his calculations. He may also do his multiplications and additions on a "desk machine," but this is not important.

If we use the above explanation as a definition we shall be in danger of circularity of argument. We avoid this by giving an outline of the means by which the desired effect is achieved. A digital computer can usually be regarded as consisting of three parts:

(i) Store.
(ii) Executive unit.
(iii) Control.

The store is a store of information, and corresponds to the human computer's paper, whether this is the paper on which he does his calculations or that on which his book of rules is printed. In so far as the human computer does calculations in his head a part of the store will correspond to his memory.

The executive unit is the part which carries out the various individual operations involved in a calculation. What these individual operations are will vary from machine to machine. Usually fairly lengthy operations can be done such as "Multiply 3540675445 by 7076345687" but in some machines only very simple ones such as "Write down 0" are possible.

We have mentioned that the "book of rules" supplied to the computer is replaced in the machine by a part of the store. It is then called the "table of instructions." It is the duty of the control to see that these instructions are obeyed correctly and in the right order. The control is so constructed that this necessarily happens.

The information in the store is usually broken up into packets of moderately small size. In one machine, for instance, a packet might consist of ten decimal digits. Numbers are assigned to the parts of the store in which the various packets of information are stored, in some systematic manner. A typical instruction might say—

"Add the number stored in position 6809 to that in 4302 and put the result back into the latter storage position."

Needless to say it would not occur in the machine expressed in English. It would more likely be coded in a form such as 6809430217. Here 17 says which of various possible operations is to be performed on the two numbers. In this case the operation is that described above, *viz.*, "Add the number. . . ." It will be noticed that the instruction takes up 10 digits and so forms one packet of information, very conveniently. The control will normally take the instructions to be obeyed in the order of the positions in which they are stored, but occasionally an instruction such as

"Now obey the instruction stored in position 5606, and continue from there"

may be encountered, or again

"If position 4505 contains 0 obey next the instruction stored in 6707, otherwise continue straight on."

Instructions of these latter types are very important because they make it possible for a sequence of operations to be replaced over and over again until some condition is fulfilled, but in doing so to obey, not fresh instructions on each repetition, but the same ones over and over again. To take a domestic analogy. Suppose Mother wants Tommy to call at the cobbler's every morning on his way to school to see if her shoes are done, she can ask him afresh every morning. Alternatively she can stick up a notice once and for all in the hall which he will see when he leaves for school and which tells him to call for the shoes, and also to destroy the notice when he comes back if he has the shoes with him.

The reader must accept it as a fact that digital computers can be constructed, and indeed have been constructed, according to the principles we have described, and that they can in fact mimic the actions of a human computer very closely.

The book of rules which we have described our human computer as using is of course a convenient fiction. Actual human computers really remember what they have got to do. If one wants to make a machine mimic the behaviour of the human computer in some complex operation one has to ask him how it is done, and then translate the answer into the form of an instruction table. Constructing instruction tables is usually described as "programming." To "programme a machine to carry out the operation A" means to put the appropriate instruction table into the machine so that it will do A.

An interesting variant on the idea of a digital computer is a "digital computer with a random element." These have instructions involving the throwing of a die or some equivalent electronic process; one such instruction might for instance be, "Throw the die and put the resulting number into store 1000." Sometimes such a machine is described as having free will (though I would not use this phrase myself). It is not normally possible to determine from observing a machine whether it has a random element, for a similar effect can be produced by such devices as making the choices depend on the digits of the decimal for π.

Most actual digital computers have only a finite store. There is no theoretical difficulty in the idea of a computer with an unlimited store. Of course only a finite part can have been used at any one time. Likewise only a finite amount can have been constructed, but we can imagine more and more being added as required. Such computers have special theoretical interest and will be called infinitive capacity computers.

The idea of a digital computer is an old one. Charles Babbage, Lucasian Professor of Mathematics at Cambridge from 1828 to 1839, planned such a machine, called the Analytical Engine, but it was never completed. Although Babbage had all the essential ideas, his machine was not at that time such a very attractive prospect. The speed which would have been available would be definitely faster than a human computer but something like 100 times slower than the Manchester machine, itself one of the slower of the modern machines. The storage was to be purely mechanical, using wheels and cards.

The fact that Babbage's Analytical Engine was to be entirely mechanical will help us to rid ourselves of a superstition. Importance is often attached to the fact that modern digital computers are electrical, and that the nervous system also is electrical. Since Babbage's machine was not electrical, and since all digital computers are in a sense equivalent, we see that this use of electricity cannot be of theoretical importance. Of course electricity usually comes in where fast signalling is concerned, so that it is not surprising that we find it in both these connections. In the nervous system chemical phenomena are at least as important as electrical. In certain computers the storage system is mainly acoustic. The feature of using electricity is thus seen to be only a very superficial similarity. If we wish to find such similarities we should look rather for mathematical analogies of function.

5. *Universality of Digital Computers*

The digital computers considered in the last section may be classified amongst the "discrete-state machines." These are the machines which move by sudden jumps or clicks from one quite definite state to another. These states are sufficiently different for the possibility of confusion between them to be ignored. Strictly speaking there are no such machines. Everything really moves continuously. But there are many kinds of machine which can profitably be *thought of* as being discrete-state machines. For instance in considering the switches for a lighting system it is a convenient fiction that each switch must be definitely on or definitely off. There must be intermediate positions, but for most purposes we can forget about them. As an example of a discrete-state machine we might consider a wheel which clicks round through 120° once a second, but may be stopped by a lever which can be operated from outside; in addition a lamp is to light in one of the positions of the wheel. This machine could be described abstractly as follows. The internal state of the machine (which is described by the position of the wheel) may be q_1, q_2 or q_3. There is an input signal i_0 or i_1 (position of lever). The internal state at any moment is determined by the last state and input signal according to the table

		Last State		
		q_1	q_2	q_3
Input	i_0	q_2	q_3	q_1
	i_1	q_1	q_2	q_3

The output signals, the only externally visible indication of the internal state (the light) are described by the table

State	q_1	q_2	q_3
Output	o_0	o_0	o_1

This example is typical of discrete-state machines. They can be described by such tables provided they have only a finite number of possible states.

It will seem that given the initial state of the machine and the input signals it is always possible to predict all future states. This is reminiscent of Laplace's view that from the complete state of the universe at one moment of time, as described by the positions and velocities of all particles, it should be possible to predict all future states. The prediction which we are considering is, however, rather nearer to practicability than that considered by Laplace. The system of the "universe as a whole" is such that quite small errors in the initial conditions can have an overwhelming effect at a later time. The displacement of a single electron by a billionth of a centimetre at one moment might make the difference between a man being killed by an avalanche a year later, or escaping. It is an essential property of the mechanical systems which we have called "discrete-state machines" that this phenomenon does not occur. Even when we consider the actual physical machines instead of the idealised machines, reasonably accurate knowledge of the state at one moment yields reasonably accurate knowledge any number of steps later.

As we have mentioned, digital computers fall within the class of discrete-state machines. But the number of states of which such a machine is capable is usually enormously large. For instance, the number for the machine now working at Manchester is about $2^{165,000}$, *i.e.*, about $10^{50,000}$. Compare this with our example of the clicking wheel described above, which had three states. It is not difficult to see why the number of states should be so immense. The computer includes a store corresponding to the paper used by a human computer. It must be possible to write into the store any one of the combinations of symbols which might have been written on the paper. For simplicity suppose that only digits from 0 to 9 are used as symbols. Variations in handwriting are ignored. Suppose the computer is allowed 100 sheets of paper each containing 50 lines each with room for 30 digits. Then the number of states is $10^{100 \times 50 \times 30}$, *i.e.*, $10^{150,000}$. This is about the number of states of three Manchester machines put together. The logarithm to the base two of the number of states is usually called the "storage capacity" of the machine. Thus the Manchester machine has a storage capacity of about 165,000 and the wheel machine of our example about 1.6. If two machines are put together their capacities must be added to obtain the capacity of the resultant machine. This leads to the possibility of statements such as "The Manchester machine contains 64 magnetic tracks each with a capacity of 2560, eight electronic tubes with a capacity of 1280. Miscellaneous storage amounts to about 300 making a total of 174,380."

Given the table corresponding to a discrete-state machine it is possible to predict what it will do. There is no reason why this calculation should not be carried out by means of a digital computer. Provided it could be carried out sufficiently quickly the digital computer could mimic the behavior of any discrete-state machine. The imitation game could then be played with the machine in question (as B) and the mimicking digital computer (as A) and the interrogator would be unable to distinguish them. Of course the digital computer must have an adequate storage capacity as well as working sufficiently fast. Moreover, it must be programmed afresh for each new machine which it is desired to mimic.

This special property of digital computers, that they can mimic any discrete-state machine, is described by saying that they are *universal* machines. The existence of machines with this property has the important consequence that, considerations of speed apart, it is unnecessary to design various new machines to do various computing processes. They can all be done with one digital computer, suitably programmed for each case. It will be seen that as a consequence of this all digital computers are in a sense equivalent.

We may now consider again the point raised at the end of §3. It was suggested tentatively that the question, "Can machines think?" should be replaced by "Are there imaginable digital computers which would do well in the imitation game?" If we wish we can make this superficially more general and ask "Are there discrete-state machines which would do well?" But in view of the universality property we see that either of these questions is equivalent to this, "Let us fix our attention on one particular digital computer *C*. Is it true that by modifying this computer to have an adequate storage, suitably increasing its speed of action, and providing it with an appropriate programme, *C* can be made to play satisfactorily the part of A in the imitation game, the part of B being taken by a man?"

6. *Contrary Views on the Main Question*

We may now consider the ground to have been cleared and we are ready to proceed to the debate on our question, "Can machines think?" and the variant of it quoted at the end of the last section. We cannot altogether abandon the original form of the problem, for opinions will differ as to the appropriateness of the substitution and we must at least listen to what has to be said in this connexion.

It will simplify matters for the reader if I explain first my own beliefs in the matter. Consider first the more accurate form of the question. I believe that in about fifty years' time it will be possible to programme computers, with a storage capacity of about 10^9, to make them play the imitation game so well that an average interrogator will not have more than 70 per cent chance of making the right identification after five minutes of questioning. The original question, "Can machines think?" I believe to be too meaningless to deserve discussion. Nevertheless I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted. I believe further that no useful purpose is served by concealing these beliefs. The popular view that scientists proceed inexorably from well-established fact to well-established fact, never being influenced by any improved conjecture, is quite mistaken. Provided it is made clear which are proved facts and which are conjectures, no harm can result. Conjectures are of great importance since they suggest useful lines of research.

I now proceed to consider opinions opposed to my own.

(1) The Theological Objection

Thinking is a function of man's immortal soul. God has given an immortal soul to every man and woman, but not to any other animal or to machines. Hence no animal or machine can think.[1]

I am unable to accept any part of this, but will attempt to reply in theological terms. I should find the argument more convincing if animals were classed with men, for there is a greater difference, to my mind, between the typical animate and the inanimate than there is between man and the other animals. The arbitrary character of the orthodox view becomes clearer if we consider how it might appear to a member of some other religious community. How do Christians regard the Moslem view that women have no souls? But let us leave this point aside and return to the main argument. It appears to me that the argument quoted above implies a serious restriction of the omnipotence of the Almighty. It is admitted that there are certain things that He cannot do such as making one equal to two, but should we not believe that He has freedom to confer a soul on an elephant if He sees fit? We might expect that He would only exercise this power in conjunction with a mutation which provided the elephant with an appropriately improved brain to minister to the needs of this soul. An argument of exactly similar form may be made for the case of machines. It may seem different because it is more difficult to "swallow." But this really only means that we think it would be less likely that He would consider the circumstances suitable for conferring a soul. The circumstances in question are discussed in the rest of this paper. In attempting to construct such machines we should not be irreverently usurping His power of creating souls, any more than we are in the procreation of children: rather we are, in either case, instruments of His will providing mansions for the souls that He creates.

However, this is mere speculation. I am not very impressed with theological arguments whatever they may be used to support. Such arguments have often been found unsatisfactory in the past. In the time of Galileo it was argued that the texts, "And the sun stood still . . . and hasted not to go down about a whole day" (Joshua x. 13) and "He laid the foundations of the earth, that it should not move at any time" (Psalm cv. 5) were an adequate refutation of the Copernican theory. With our present knowledge such an argument appears futile. When that knowledge was not available it made a quite different impression.

[1] Possibly this view is heretical. St. Thomas Aquinas [*Summa Theologica,* quoted by Bertrand Russell (1945, p. 458)] states that God cannot make a man to have no soul. But this may not be a real restriction on His powers, but only a result of the fact that men's souls are immortal, and therefore indestructible.

(2) The "Heads in the Sand" Objection

"The consequences of machines thinking would be too dreadful. Let us hope and believe that they cannot do so."

This argument is seldom expressed quite so openly as in the form above. But it affects most of us who think about it at all. We like to believe that Man is in some subtle way superior to the rest of creation. It is best if he can be shown to be *necessarily* superior, for then there is no danger of him losing his commanding position. The popularity of the theological argument is clearly connected with this feeling. It is likely to be quite strong in intellectual people, since they value the power of thinking more highly than others, and are more inclined to base their belief in the superiority of Man on this power.

I do not think that this argument is sufficiently substantial to require refutation. Consolation would be more appropriate: perhaps this should be sought in the transmigration of souls.

(3) The Mathematical Objection

There are a number of results of mathematical logic which can be used to show that there are limitations to the powers of discrete-state machines. The best known of these results is known as Gödel's theorem (1931) and shows that in any sufficiently powerful logical system statements can be formulated which can neither be proved nor disproved within the system, unless possibly the system itself is inconsistent. There are other, in some respects similar, results due to Church (1936), Kleene (1935), Rosser, and Turing (1937). The latter result is the most convenient to consider, since it refers directly to machines, whereas the others can only be used in a comparatively indirect argument: for instance if Gödel's theorem is to be used we need in addition to have some means of describing logical systems in terms of machines, and machines in terms of logical systems. The result in question refers to a type of machine which is essentially a digital computer with an infinite capacity. It states that there are certain things that such a machine cannot do. If it is rigged up to give answers to questions as in the imitation game, there will be some questions to which it will either give a wrong answer, or fail to give an answer at all however much time is allowed for a reply. There may, of course, be many such questions, and questions which cannot be answered by one machine may be satisfactorily answered by another. We are of course supposing for the present that the questions are of the kind to which an answer "Yes" or "No" is appropriate, rather than questions such as "What do you think of Picasso?" The questions that we know the machines must fail on are of this type, "Consider the machine specified as follows. . . . Will this machine ever answer 'Yes' to any question?" The dots are to be replaced by a description of some machine in a standard form, which could be something like that used in §5. When the machine described bears a certain comparatively simple relation to the machine which is under interrogation, it can be shown that the answer is either wrong or not forthcoming. This is the mathematical result: it is argued that it proves a disability of machines to which the human intellect is not subject.

The short answer to this argument is that although it is established that there are limitations to the powers of any particular machine, it has only been stated, without any sort of proof, that no such limitations apply to the human intellect. But I do not think this view can be dismissed quite so lightly. Whenever one of these machines is asked the appropriate critical question, and gives a definite answer, we know that this answer must be wrong, and this gives us a certain feeling of superiority. Is this feeling illusory? It is no doubt quite genuine, but I do not think too much importance should be attached to it. We too often give wrong answers to questions ourselves to be justified in being very pleased at such evidence of fallibility on the part of the machines. Further, our superiority can only be felt on such an occasion in relation to the one machine over which we have scored our petty triumph. There would be no question of triumphing simultaneously over *all* machines. In short, then, there might be men cleverer than any given machine, but then again there might be other machines cleverer again, and so on.

Those who hold to the mathematical argument would, I think, mostly be willing to accept the imitation game as a basis for discussion. Those who believe in the two previous objections would probably not be interested in any criteria.

(4) The Argument from Consciousness

This argument is very well expressed in Professor Jefferson's Lister Oration for 1949, from which I quote. "Not until a machine can write a sonnet or compose a concerto because of thoughts and emotions felt, and not by the chance fall of symbols, could we agree that machine equals brain—that is, not only write it but know that it had written it. No mechanism could feel (and not merely artificially signal, an easy contrivance) pleasure at its successes, grief when its valves fuse, be warmed by flattery, be made miserable by its mistakes, be charmed by sex, be angry or depressed when it cannot get what it wants."

This argument appears to be a denial of the validity of our test. According to the most extreme form of this view the only way by which one could be sure that a machine thinks is to *be* the machine and to feel oneself thinking. One could then describe these feelings to the world, but of course no one would be justified in taking any notice. Likewise according to this view the only way to know that a *man* thinks is to be that particular man. It is in fact the solipsist point of view. It may be the most logical view to hold but it makes communication of ideas difficult. A is liable to believe "A thinks but B does not" whilst B believes "B thinks but A does not." Instead of arguing continually over this point it is usual to have the polite convention that everyone thinks.

I am sure that Professor Jefferson does not wish to adopt the extreme and solipsist point of view. Probably he would be quite willing to accept the imitation game as a test. The game (with the player B omitted) is frequently used in practice under the name of *viva voce* to discover whether some one really understands something or has "learnt it parrot fashion." Let us listen in to a part of such a *viva voce:*

Interrogator: In the first line of your sonnet which reads "Shall I compare thee to a summer's day," would not "a spring day" do as well or better?
Witness: It wouldn't scan.
Interrogator: How about "a winter's day." That would scan all right.
Witness: Yes, but nobody wants to be compared to a winter's day.
Interrogator: Would you say Mr. Pickwick reminded you of Christmas?
Witness: In a way.
Interrogator: Yet Chrismas is a winter's day, and I do not think Mr. Pickwick would mind the comparison.
Witness: I don't think you're serious. By a winter's day one means a typical winter's day, rather than a special one like Christmas.

And so on. What would Professor Jefferson say if the sonnet-writing machine was able to answer like this in the *viva voce?* I do not know whether he would regard the machine as "merely artificially signalling" these answers, but if the answers were as satisfactory and sustained as in the above passage I do not think he would describe it as "an easy contrivance." This phrase is, I think, intended to cover such devices as the inclusion in the machine of a record of someone reading a sonnet, with appropriate switching to turn it on from time to time.

In short then, I think that most of those who support the argument from consciousness could be persuaded to abandon it rather than be forced into the solipsist position. They will then probably be willing to accept our test.

I do not wish to give the impression that I think there is no mystery about consciousness. There is, for instance, something of a paradox connected with any attempt to localise it. But I do not think these mysteries necessarily need to be solved before we can answer the question with which we are concerned in this paper.

(5) Arguments from Various Disabilities

These arguments take the form, "I grant you that you can make machines do all the things you have mentioned but you will never be able to make one to do X." Numerous features X are suggested in this connexion. I offer a selection:

Be kind, resourceful, beautiful, friendly, have initiative, have a sense of humour, tell right from wrong, make mistakes, fall in love, enjoy strawberries and cream, make some one fall in love with it, learn from experience, use words properly, be the subject of its own thought, have as much diversity of behaviour as a man, do something really new.

No support is usually offered for these statements. I believe they are mostly founded on the principle of scientific induction. A man has seen thousands of machines in his lifetime. From what he sees of them he draws a number of general conclusions. They are ugly, each is designed for a very limited purpose, when required for a minutely different purpose they are useless, the variety of behaviour of any one of them is very small, etc., etc. Naturally he concludes that these are necessary properties of machines in general. Many of these limitations are associated with the very small storage capacity of most machines. (I am assuming that the idea of storage capacity is extended in some way to cover machines other than discrete-state machines. The exact definition does not matter as no mathematical accuracy is claimed in the present discussion.) A few years ago, when very little had been heard of digital computers, it was possible to elicit much incredulity concerning them, if one mentioned their properties without describing their construction. That was presumably due to a similar application of the principle of scientific induction. These applications of the principle are of course largely unconscious. When a burnt child fears the fire and shows that he fears it by avoiding it, I should say that he was applying scientific induction. (I could of course also describe his behaviour in many other ways.) The works and customs of mankind do not seem to be very suitable material to which to apply scientific induction. A very large part of space-time must be investigated, if reliable results are to be obtained. Otherwise we may (as most English children do) decide that everybody speaks English, and that it is silly to learn French.

There are, however, special remarks to be made about many of the disabilities that have been mentioned. The inability to enjoy strawberries and cream may have struck the reader as frivolous. Possibly a machine might be made to enjoy this delicious dish, but any attempt to make one do so would be idiotic. What is important about this disability is that it contributes to some of the other disabilities, *e.g.*, to the difficulty of the same kind of friendliness occurring between man and machine as between white man and white man, or between black man and black man.

The claim that "machines cannot make mistakes" seems a curious one. One is tempted to retort, "Are they any the worse for that?" But let us adopt a more sympathetic attitude, and try to see what is really meant. I think this criticism can be explained in terms of the imitation game. It is claimed that the interrogator could distinguish the machine from the man simply by setting them a number of problems in arithmetic. The machine would be unmasked because of its deadly accuracy. The reply to this is simple. The machine (programmed for playing the game) would not attempt to give the *right* answers to the arithmetic problems. It would deliberately introduce mistakes in a manner calculated to confuse the interrogator. A mechanical fault would probably show itself through an unsuitable decision as to what sort of a mistake to make in the arithmetic. Even this interpretation of the criticism is not sufficiently sympathetic. But we cannot afford the space to go into it much further. It seems to me that this criticism depends on a confusion between two kinds of mistake. We may call them "errors of functioning" and "errors of conclusion." Errors of functioning are due to some mechanical or electrical fault which causes the machine to behave otherwise than it was designed to do. In philosophical discussions one likes to ignore the possibility of such errors; one is therefore discussing "abstract machines." These abstract machines are mathematical fictions rather than physical objects. By definition they are incapable of errors of functioning. In this sense we can truly say that "machines can never make mistakes." Errors of conclusion can only arise when some meaning is attached to the output signals from the machine. The machine might, for instance, type out mathematical equations, or sentences in English. When a false proposition is typed we say that the machine has committed an error of conclusion. There is clearly no reason at all for saying that a machine cannot make this kind of mistake. It might do nothing but type out repeatedly "$0 = 1$." To take a less perverse example, it might have some method for drawing conclusions by scientific induction. We must expect such a method to lead occasionally to erroneous results.

The claim that a machine cannot be the subject of its own thought can of course only be answered if it can be shown that the machine has *some* thought with *some* subject matter. Nevertheless, "the subject matter of a machine's operations" does seem to mean something, at least to the people who deal with it. If, for instance, the machine was trying to find a solution of the equation $x^2 - 40x - 11 = 0$ one would be tempted to describe this equation as part of the machine's subject matter at that moment. In this sort of sense a machine undoubtedly can be its own subject matter. It may be used to help in making up its own programmes, or to predict the effect of alterations in its own structure. By observing the results of its own behaviour it can modify its own programmes so as to achieve some purpose more effectively. These are possibilities of the near future, rather than Utopian dreams.

The criticism that a machine cannot have much diversity of behaviour is just a way of saying that it cannot have much storage capacity. Until fairly recently a storage capacity of even a thousand digits was very rare.

The criticisms that we are considering here are often disguised forms of the argument from consciousness. Usually if one maintains that a machine *can* do one of these things, and describes the kind of method that the machine could use, one will not make much of an impression. It is thought that the method (whatever it may be, for it must be mechanical) is really rather base. Compare the parentheses in Jefferson's statement quoted on page 22.

(6) Lady Lovelace's Objection

Our most detailed information of Babbage's Analytical Engine comes from a memoir by Lady Lovelace (1842). In it she states, "The Analytical Engine has no pretensions to *originate* anything. It can do *whatever we know how to order it* to perform" (her italics). This statement is quoted by Hartree (1949) who adds: "This does not imply that it may not be possible to construct electronic equipment which will 'think for itself,' or in which, in biological terms, one could set up a conditioned reflex, which would serve as a basis for 'learning.' Whether this is possible in principle or not is a stimulating and exciting question, suggested by some of these recent developments. But it did not seem that the machines constructed or projected at the time had this property."

I am in thorough agreement with Hartree over this. It will be noticed that he does not assert that the machines in question had not got the property, but rather that the evidence available to Lady Lovelace did not encourage her to believe that they had it. It is quite possible that the machines in question had in a sense got this property. For suppose that some discrete-state machine has the property. The Analytical Engine was a universal digital computer, so that, if its storage capacity and speed were adequate, it could by suitable programming be made to mimic the machine in question. Probably this argument did not occur to the Countess or to Babbage. In any case there was no obligation on them to claim all that could be claimed.

This whole question will be considered again under the heading of learning machines.

A variant of Lady Lovelace's objection states that a machine can "never do anything really new." This may be parried for a moment with the saw, "There is nothing new under the sun." Who can be certain that "original work" that he has done was not simply the growth of the seed planted in him by teaching, or the effect of following well-known general principles. A better variant of the objection says that a machine can never "take us by surprise." This statement is a more direct challenge and can be met directly. Machines take me by surprise with great frequency. This is largely because I do not do sufficient calculation to decide what to expect them to do, or rather because, although I do a calculation, I do it in a hurried, slipshod fashion, taking risks. Perhaps I say to myself, "I suppose the voltage here ought to be the same as there: anyway let's assume it is." Naturally I am often wrong, and the result is a surprise for me for by the time the experiment is done these assumptions have been forgotten. These admissions lay me open to lectures on the subject of my vicious ways, but do not throw any doubt on my credibility when I testify to the surprises I experience.

I do not expect this reply to silence my critic. He will probably say that such surprises are due to some creative mental act on my part, and reflect no credit on the machine. This leads us back to the argument from consciousness, and far from the idea of surprise. It is a line of argument we must consider closed, but it is perhaps worth remarking that the appreciation of something as surprising requires as much of a "creative mental act" whether the surprising event originates from a man, a book, a machine or anything else.

The view that machines cannot give rise to surprises is due, I believe, to a fallacy to which philosophers and mathematicians are particularly subject. This is the assumption that as soon as a fact is presented to a mind all consequences of that fact spring into the mind simultaneously with it. It is a very useful assumption under many circumstances, but one too easily forgets that it is false. A natural consequence of doing so is that one then assumes that there is no virtue in the mere working out of consequences from data and general principles.

(7) Argument from Continuity in the Nervous System

The nervous system is certainly not a discrete-state machine. A small error in the information about the size of a nervous impulse impinging on a neuron, may make a large difference to the size of the outgoing impulse. It may be argued that, this being so, one cannot expect to be able to mimic the behaviour of the nervous system with a discrete-state system.

It is true that a discrete-state machine must be different from a continuous machine. But if we adhere to the conditions of the imitation game, the interrogator will not be able to take any advantage of this difference. The situation can be made clearer if we consider some other simpler continuous machine. A differential analyser will do very well. (A differential analyser is a certain kind of machine not of the discrete-state type used for some kinds of calculation.) Some of these provide their answers in a typed form, and so are suitable for taking part in the game. It would not be possible for a digital computer to predict exactly what answers the differential analyser would give to a problem, but it would be quite capable of giving the right sort of answer. For instance, if asked to give the value of π (actually about 3.1416) it would be reasonable to choose at random between the values 3.12, 3.13, 3.14, 3.15, 3.16 with the probabilities of 0.05, 0.15, 0.55, 0.19, 0.06 (say). Under these circumstances it would be very difficult for the interrogator to distinguish the differential analyser from the digital computer.

(8) The Argument from Informality of Behaviour

It is not possible to produce a set of rules purporting to describe what a man should do in every conceivable set of circumstances. One might for instance have a rule that one is to stop when one sees a red traffic light, and to go if one sees a green one, but what if by some fault both appear together? One may perhaps decide that it is safest to stop. But some further difficulty may well arise from this decision later. To attempt to provide rules of conduct to cover every eventuality, even those arising from traffic lights, appears to be impossible. With all this I agree.

From this it is argued that we cannot be machines. I shall try to reproduce the argument, but I fear I shall hardly do it justice. It seems to run something like this. "If each man had a definite set of rules of conduct by which he regulated his life he would be no better than a machine. But there are no such rules, so men cannot be machines." The undistributed middle is glaring. I do not think the argument is ever put quite like this, but I believe this is the argument used nevertheless. There may however be a certain confusion between "rules of conduct" and "laws of behaviour" to cloud the issue. By "rules of conduct" I mean precepts such as "Stop if you see red lights," on which one can act, and of which one can be conscious. By "laws of behaviour" I mean laws of nature as applied to a man's body such as "if you pinch him he will squeak." If we substitute "laws of behaviour which regulate his life" for "laws of conduct by which he regulates his life" in the argument quoted the undistributed middle is no longer insuperable. For we believe that it is not only true that being regulated by laws of behaviour implies being some sort of machine (though not necessarily a discrete-state machine), but that conversely being such a machine implies being regulated by such laws. However, we cannot so easily convince ourselves of the absence of complete laws of behaviour as of complete rules of conduct. The only way we know of for finding such laws is scientific observation, and we certainly know of no circumstances under which we could say, "We have searched enough. There are no such laws."

We can demonstrate more forcibly that any such statement would be unjustified. For suppose we could be sure of finding such laws if they existed. Then given a discrete-state machine it should certainly be possible to discover by observation sufficient about it to predict its future behaviour, and this within a reasonable time, say a thousand years. But this does not seem to be the case. I have set up on the Manchester computer a small programme using only 1,000 units of storage, whereby the machine supplied with one sixteen-figure number replies with another within two seconds. I would defy anyone to learn from these replies sufficient about the programme to be able to predict any replies to untried values.

(9) The Argument from Extrasensory Perception

I assume that the reader is familiar with the idea of extrasensory perception, and the meaning of the four items of it, *viz.*, telepathy, clairvoyance, precognition and psychokinesis. These disturbing phenomena seem to deny all our usual scientific ideas. How we should like to discredit them! Unfortunately the statistical evidence, at least for telepathy, is overwhelming. It is very difficult to rearrange one's ideas so as to fit these new facts in. Once one has accepted them it does not seem a very big step to believe in ghosts and bogies. The idea that our bodies move simply according to the known laws of physics, together with some others not yet discovered but somewhat similar, would be one of the first to go.

This argument is to my mind quite a strong one. One can say in reply that many scientific theories seem to remain workable in practice, in spite of clashing with ESP; that in fact one can get along very nicely if one forgets about it. This is rather cold comfort, and one fears that thinking is just the kind of phenomenon where ESP may be especially relevant.

A more specific argument based on ESP might run as follows: "Let us play the imitation game, using as witnesses a man who is good as a telepathic receiver, and a digital computer. The interrogator can ask such questions as 'What suit does the card in my right hand belong to?' The man by telepathy or clairvoyance gives the right answer 130 times out of 400 cards. The machine can only guess at random, and perhaps gets 104 right, so the interrogator makes the right identification." There is an interesting possibility which opens here. Suppose the digital computer contains a random number generator. Then it will be natural to use this to decide what answer to give. But then the random number generator will be subject to the psychokinetic powers of the interrogator. Perhaps this psychokinesis might cause the machine to guess right more often than would be expected on a probability calculation, so that the interrogator might still be unable to make the right identification. On the other hand, he might be able to guess right without any questioning, by clairvoyance. With ESP anything may happen.

If telepathy is admitted it will be necessary to tighten our test up. The situation could be regarded as analogous to that which would occur if the interrogator were talking to himself and one of the competitors was listening with his ear to the wall. To put the competitors into a "telepathy-proof room" would satisfy all requirements.

7. Learning Machines

The reader will have anticipated that I have no very convincing arguments of a positive nature to support my views. If I had I should not have taken such pains to point out the fallacies in contrary views. Such evidence as I have I shall now give.

Let us return for a moment to Lady Lovelace's objection, which stated that the machine can only do what we tell it to do. One could say that a man can "inject" an idea into the machine, and that it will respond to a certain extent and then drop into quiescence, like a piano string struck by a hammer. Another simile would be an atomic pile of less than critical size: an injected idea is to correspond to a neutron entering the pile from without. Each such neutron will cause a certain disturbance which eventually dies away. If, however, the size of the pile is sufficiently increased, the disturbance caused by such an incoming neutron will very likely go on and on increasing until the whole pile is destroyed. Is there a corresponding phenomenon for minds, and is there one for machines? There does seem to be one for the human mind. The majority of them seem to be "subcritical," *i.e.*, to correspond in this analogy to piles of subcritical size. An idea presented to such a mind will on average give rise to less than one idea in reply. A smallish proportion are supercritical. An idea presented to such a mind that may give rise to a whole "theory" consisting of secondary, tertiary and more remote ideas. Animals minds seem to be very definitely subcritical. Adhering to this analogy we ask, "Can a machine be made to be supercritical?"

The "skin-of-an-onion" analogy is also helpful. In considering the functions of the mind or the brain we find certain operations which we can explain in purely mechanical terms. This we say does not correspond to the real mind: it is a sort of skin which we must strip off if we are to find the real mind. But then in what remains we find a further skin to be stripped off, and so on. Proceeding in this way do we ever come to the "real" mind, or do we eventually come to the skin which has nothing in it? In the latter case the whole mind is mechanical. (It would not be a discrete-state machine however. We have discussed this.)

These last two paragraphs do not claim to be convincing arguments. They should rather be described as "recitations tending to produce belief."

The only really satisfactory support that can be given for the view expressed at the beginning of §6, will be that provided by waiting for the end of the century and then doing the experiment described. But what can we say in the meantime? What steps should be taken now if the experiment is to be successful?

As I have explained, the problem is mainly one of programming. Advances in engineering will have to be made too, but it seems unlikely that these will not be adequate for the requirements. Estimates of the storage capacity of the brain vary from 10^{10} to 10^{15} binary digits. I incline to the lower values and believe that only a very small fraction is used for the higher types of thinking. Most of it is probably used for the retention of visual impressions. I should be surprised if more than 10^9 was required for satisfactory playing of the imitation game, at any rate against a blind man. (*Note:* The capacity of the *Encyclopaedia Britannica,* 11th edition, is 2×10^9.) A storage capacity of 10^7 would be a very practicable possibility even by present techniques. It is probably not necessary to increase the speed of operations of the machines at all. Parts of modern machines which can be regarded as analogs of nerve cells work about a thousand times faster than the latter. This should provide a "margin of safety" which could cover losses of speed arising in many ways. Our problem then is to find out how to programme these machines to play the game. At my present rate of working I produce about a thousand digits of programme a day, so that about sixty workers, working steadily through the fifty years might accomplish the job, if nothing went into the wastepaper basket. Some more expeditious method seems desirable.

In the process of trying to imitate an adult human mind we are bound to think a good deal about the process which has brought it to the state that it is in. We may notice three components.

(*a*) The initial state of the mind, say at birth,
(*b*) The education to which it has been subjected,
(*c*) Other experience, not to be described as education, to which it has been subjected.

Instead of trying to produce a programme to simulate the adult mind, why not rather try to produce one which simulates the child's? If this were then subjected to an appropriate course of education one would obtain the adult brain. Presumably the child brain is something like a notebook as one buys it from the stationer's. Rather little mechanism, and lots of blank sheets. (Mechanism and writing are from our point of view almost synonymous.) Our hope is that there is so little mechanism in the child brain that something like it can be easily programmed. The amount of work in the education we can assume, as a first approximation, to be much the same as for the human child.

We have thus divided our problem into two parts. The child programme and the education process. These two remain very closely connected. We cannot expect to find a good child machine at the first attempt. One must experiment with teaching one such machine and see how well it learns. One can then try another and see if it is better or worse. There is an obvious connection between this process and evolution, by the identifications

Structure of the child machine	=	hereditary material
Changes of the child machine	=	mutations
Natural selection	=	judgment of the experimenter

One may hope, however, that this process will be more expeditious than evolution. The survival of the fittest is a slow method for measuring advantages. The experimenter, by the exercise of intelligence, should be able to speed it up. Equally important is the fact that he is not restricted to random mutations. If he can trace a cause for some weakness he can probably think of the kind of mutation which will improve it.

It will not be possible to apply exactly the same teaching process to the machine as to a normal child. It will not, for instance, be provided with legs, so that it could not be asked to go out and fill the coal scuttle. Possibly it might not have eyes. But however well these deficiencies might be overcome by clever engineering, one could not send the creature to school without the other children making excessive fun of it. It must be given some tuition. We need not be too concerned about the legs, eyes, etc. The example of Miss Helen Keller shows that education can take place provided that communication in both directions between teacher and pupil can take place by some means or other.

We normally associate punishments and rewards with the teaching process. Some simple child machines can be constructed or programmed on this sort of principle. The machine has to be so constructed that events which shortly preceded the occurrence of a punishment signal are unlikely to be repeated, whereas a reward signal increased the probability of repetition of the events which led up to it. These definitions do not presuppose any feelings on the part of the machine. I have done some experiments with one such child machine, and succeeded in teaching it a few things, but the teaching method was too unorthodox for the experiment to be considered really successful.

The use of punishments and rewards can at best be a part of the teaching process. Roughly speaking, if the teacher has no other means of communicating to the pupil, the amount of information which can reach him does not exceed the total number of rewards and punishments applied. By the time a child has learnt to repeat "Casabianca" he would probably feel very sore indeed, if the text could only be discovered by a "Twenty Questions" technique, every "NO" taking the form of a blow. It is necessary therefore to have some other "unemotional" channels of communication. If these are available it is possible to teach a machine by punishments and rewards to obey orders given in some language, *e.g.*, a symbolic language. These orders are to be transmitted through the "unemotional" channels. The use of this language will diminish greatly the number of punishments and rewards required.

Opinions may vary as to the complexity which is suitable in the child machine. One might try to make it as simple as possible consistently with the general principles. Alternatively one might have a complete system of logical inference "built in."[2] In the latter case the store would be largely occupied with definitions and propositions. The propositions would have various kinds of status, *e.g.*, well-established facts, conjectures, mathematically proved theorems, statements given by an authority, expressions having the logical form of proposition but not belief-value. Certain propositions may be described as "imperatives." The machine should be so constructed that as soon as an imperative is classed as "well established" the appropriate action automatically takes place. To illustrate this, suppose the teacher says to the machine, "Do your homework now." This may cause "Teacher says 'Do your homework now' " to be included amongst the well-established facts. Another such fact might be, "Everything that teacher says is true." Combining these may eventually lead to the imperative, "Do your homework now," being included amongst the well-established facts, and this, by the construction of the machine, will mean that the homework actually gets started, but the effect is very satisfactory. The processes of inference used by the machine need not be such as would satisfy the most exacting logicians. There might for instance be no hierarchy of types. But this need not mean that type fallacies will occur, any more than we are bound to fall over unfenced cliffs. Suitable imperatives (expressed *within* the systems, not forming part of the rules *of* the system) such as "Do not use a class unless it is a subclass of one which has been mentioned by teacher" can have a similar effect to "Do not go too near the edge."

The imperatives that can be obeyed by a machine that has no limbs are bound to be of a rather intellectual character, as in the example (doing homework) given above. Important amongst such imperatives will be ones which regulate the order in which the rules of the logical system concerned are to be applied. For at each stage when one is using a logical system, there is a very large number of alternative steps, any of which one is permitted to apply, so far as obedience to the rules of the logical system is concerned. These choices make the difference between a brilliant and a footling reasoner, not the difference between a sound and a fallacious one. Propositions leading to imperatives of this kind might be "When Socrates is mentioned, use the syllogism in Barbara" or "If one method has been proved to be quicker than another, do not use the slower method." Some

[2] Or rather "programmed in" for our child machine will be programmed in a digital computer. But the logical system will not have to be learnt.

of these may be "given by authority," but others may be produced by the machine itself, *e.g.* by scientific induction.

The idea of a learning machine may appear paradoxical to some readers. How can the rules of operation of the machine change? They should describe completely how the machine will react whatever its history might be, whatever changes it might undergo. The rules are thus quite time-invariant. This is quite true. The explanation of the paradox is that the rules which get changed in the learning process are of a rather less pretentious kind, claiming only an ephemeral validity. The reader may draw a parallel with the Constitution of the United States.

An important feature of a learning machine is that its teacher will often be very largely ignorant of quite what is going on inside, although he may still be able to some extent to predict his pupil's behavior. This should apply most strongly to the later education of a machine arising from a child machine of well-tried design (or programme). This is in clear contrast with normal procedure when using a machine to do computations: one's object is then to have a clear mental picture of the state of the machine at each moment in the computation. This object can only be achieved with a struggle. The view that "the machine can only do what we know how to order it to do,"[3] appears strange in face of this. Most of the programmes which we can put into the machine will result in its doing something that we cannot make sense of at all, or which we regard as completely random behaviour. Intelligent behaviour presumably consists in a departure from the completely disciplined behaviour involved in computation, but a rather slight one, which does not give rise to random behaviour, or to pointless repetitive loops. Another important result of preparing our machine for its part in the imitation game by a process of teaching and learning is that "human fallibility" is likely to be omitted in a rather natural way, *i.e.,* without special "coaching." (The reader should reconcile this with the point of view on pages 23 and 24.) Processes that are learnt do not produce a hundred per cent certainty of result; if they did they could not be unlearnt.

It is probably wise to include a random element in a learning machine. A random element is rather useful when we are searching for a solution of some problem. Suppose for instance we wanted to find a number between 50 and 200 which was equal to the square of the sum of its digits, we might start at 51 then try 52 and go on until we got a number that worked. Alternatively we might choose numbers at random until we got a good one. This method has the advantage that it is unnecessary to keep track of the values that have been tried, but the disadvantage that one may try the same one twice, but this is not very important if there are several solutions. The systematic method has the disadvantage that there may be

[3] Compare Lady Lovelace's statement which does not contain the word "only."

an enormous block without any solutions in the region which has to be investigated first. Now the learning process may be regarded as a search for a form of behaviour which will satisfy the teacher (or some other criterion). Since there is probably a very large number of satisfactory solutions the random method seems to be better than the systematic. It should be noticed that it is used in the analogous process of evolution. But there the systematic method is not possible. How could one keep track of the different genetical combinations that had been tried, so as to avoid trying them again?

We may hope that machines will eventually compete with men in all purely intellectual fields. But which are the best ones to start with? Even this is a difficult decision. Many people think that a very abstract activity, like the playing of chess, would be best. It can also be maintained that it is best to provide the machine with the best sense organs that money can buy, and then teach it to understand and speak English. This process could follow the normal teaching of a child. Things would be pointed out and named, etc. Again I do not know what the right answer is, but I think both approaches should be tried.

We can only see a short distance ahead, but we can see plenty there that needs to be done.

section 2

Machines That Play Games

A favorite area of research in artificial intelligence, past and present, is in computer programs that play games. Why should one be interested in game playing, a mere human pastime? Or, as a Soviet acquaintance once put the question to one of the editors of this volume, "Who allows you to do it?"

Game playing has many fascinating aspects to the researcher. Affectively, it provides a direct contest between man's wit and machine's wit. On a more serious level, game situations provide problem environments which are relatively highly regular and well defined, but which afford sufficient complexity in solution generation so that intelligence and symbolic reasoning skills play a crucial role. In short, game environments are very useful task environments for studying the nature and structure of complex problem-solving processes.

The game of chess is one of man's valued intellectual diversions, and a number of chess-playing programs have been constructed. A history and critique of these efforts are given by Newell, Shaw, and Simon in the paper reprinted in this section. The greater part of their paper is devoted to a detailed explanation of the working of their own chess-playing program (NSS Chess Player). Following this theoretical explanation is a game played by the NSS Chess Player. It is annotated by a number of chess experts, including a partial annotation by chess master Edward Lasker.

The NSS Chess Player is one of those research efforts which lie in the shadowy area between artificial intelligence and simulation of human problem-solving. In the strict sense, it is not intended to be a model of human problem-solving in the chess environment. But the authors, in conceiving their program, were convinced that humanlike problem-solving methods, involving highly adaptive and highly selective search techniques, would be more effective in chess problem-solving than other computational schemes that had been proposed and tried. The behavior of their chess-playing program tends to support their conviction.

The focus on human problem-solving methods which is characteristic of the research of Newell, Shaw, and Simon predates the existence of their research team. In this connection, it is instructive to read the essays in Simon's *Models of Man* entitled "A Behavioral Theory of Rational Choice" and "Rational Choice and the Structure of the Environment." These essays, written more than a decade ago, contain a large part of the basic conceptual scheme of the decision-making mechanisms embodied in the Chess Player and the Logic Theorist, as well as in the portfolio selection program of Clarkson (reprinted in Part 2 of this volume).

Samuel's checker-playing program, on the other hand, sits squarely in the artificial intelligence camp. In its basic mechanism, especially in its position-evaluation scheme, it does not employ humanlike problem-solving mechanisms. Samuel believes that the effective path to progress in artificial intelligence is probably not that of imitating and adapting human processes. The rather high level of skill attained by Samuel's program is reassuring as to this point of view.

In terms of actual proficiency as exhibited in behavior, Samuel's program is one of the landmarks of artificial intelligence research to date. A recent game played by the program, in which it defeated a checkers champion, follows the Samuel article.

Of special interest in the checker-playing program are the learning routines, which improve the performance of the program as it gains experience with actual games. This learning scheme is important because it represents the only really successful attempt at machine learning in problem-solving situations.

A. Newell is Institute Professor of Systems and Communication Sciences at Carnegie Institute of Technology. H. A. Simon is Professor of Administration and Psychology in the Graduate School of Industrial Administration at the same institution. J. C. Shaw is a member of the research staff of the RAND Corporation. A. Samuel is Director of Research Communications at the IBM Research Center.

CHESS-PLAYING PROGRAMS AND THE PROBLEM OF COMPLEXITY

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

Man can solve problems without knowing how he solves them. This simple fact sets the conditions for all attempts to rationalize and understand human decision-making and problem-solving. Let us simply assume that it is good to know how to do mechanically what man can do naturally—both to add to man's knowledge of man, and to add to his kit of tools for controlling and manipulating his environment. We shall try to assess recent progress in understanding and mechanizing man's intellectual attainments by considering a single line of attack—the attempts to construct digital computer programs that play chess.

Chess is the intellectual game *par excellence.* Without a chance device to obscure the contest, it pits two intellects against each other in a situation so complex that neither can hope to understand it completely, but sufficiently amenable to analysis that each can hope to outthink his opponent. The game is sufficiently deep and subtle in its implications to have supported the rise of professional players, and to have allowed a deepening analysis through 200 years of intensive study and play without becoming exhausted or barren. Such characteristics mark chess as a natural arena for attempts at mechanization. If one could devise a successful chess machine, one would seem to have penetrated to the core of human intellectual endeavor.

The history of chess programs is an example of the attempt to conceive and cope with complex mechanisms. Now there might have been a trick—one might have discovered something that was as the wheel to the human leg: a device quite different from humans in its methods, but supremely effective in its way, and perhaps very simple. Such a device might play excellent chess, but would fail to further our understanding of human intellectual processes. Such a prize, of course, would be worthy of discovery in its own right, but there appears to be nothing of this sort in sight.

We return to the original orientation: Humans play chess, and when they do they engage in behavior that seems extremely complex, intricate, and successful. Consider, for example, a scrap of a player's (White's) running comment as he analyzes the position in Fig. 1:

> *Are there any other threats? Black also has a threat of Knight to Bishop 5 threatening the Queen, and also putting more pressure on the King's side because his Queen's Bishop can come over after he moves his Knight at Queen 2; however, that is not the immediate threat. Otherwise, his Pawn at King 4 is threatening my Pawn. . . .*

Notice that his analysis is qualitative and functional. He wanders from one feature to another, accumulating various bits of information that will be available from time to time throughout the rest of the analysis. He makes evaluations in terms of pressures and immediacies of threat, and gradually creates order out of the situation.

How can we construct mechanisms that will show comparable complexity in their behavior? They need not play in exactly the same way; close simulation of the human is not the immediate issue. But we do assert that complexity of behavior is essential to an intelligent performance—that the complexity of a successful chess program will approach the complexity

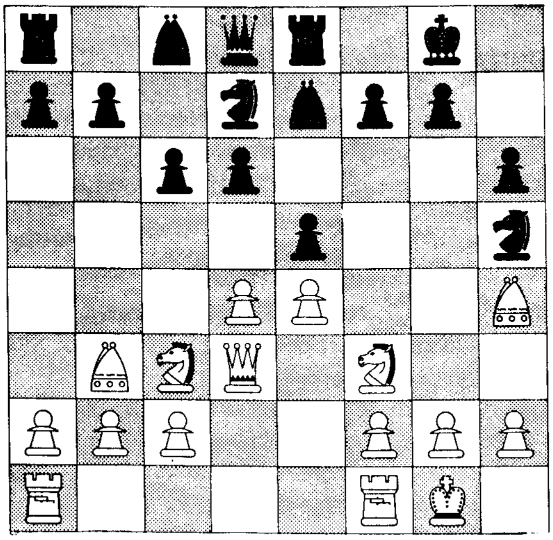


Figure 1.

of the thought processes of a successful human chess player. Complexity of response is dictated by the task, not by idiosyncrasies of the human response mechanism.

There is a close and reciprocal relation between complexity and communication. On the one hand, the complexity of the systems we can specify depends on the language in which we must specify them. Being human, we have only limited capacities for processing information. Given a more powerful language, we can specify greater complexity with limited processing powers.

Let us illustrate this side of the relation between complexity and communication. No one considers building chess machines in the literal sense—fashioning pieces of electronic gear into automatons that will play chess. We think instead of chess programs; specifications written in a language, called machine code, that will instruct a digital computer of standard design how to play chess. There is a reason for choosing this latter course—in addition to any aversion we may have to constructing a large piece of special-purpose machinery. Machine code is a more powerful language than the block diagrams of the electronics engineer. Each symbol in machine code specifies a larger unit of processing than a symbol in the block diagram. Even a moderately complicated program becomes hopelessly complex if thought of in terms of gates and pulses.

But there is another side to the relation between communication and complexity. We cannot use any old language we please. We must be understood by the person or machine to whom we are communicating. English will not do to specify chess programs because there are no English-understanding computers. A specification in English is a specification to another human who then has the task of creating the machine. Machine code is an advance precisely because there are machines that understand it—because a chess program in machine code is operationally equivalent to a machine that plays chess.

If the machine could understand even more powerful languages, we could use these to write chess programs—and thus get more complex and intelligent programs from our limited human processing capacity. But communication is limited by the intelligence of the least participant, and at present a computer has only passive capability. The language it understands is one of simple commands—it must be told very much about what to do.

Thus it seems that the rise of effective communication between man and computer will coincide with the rise in the intelligence of the computer—so that the human can say more while thinking less. But at this point in history, the only way we can obtain more intelligent machines is to design them—we cannot yet grow them, or breed them, or train them by the blind procedures that work with humans. We are caught at the wrong equilibrium of a bistable system: we could design more intelligent machines if we could communicate to them better; we could communicate to them better if they were more intelligent. Limited both in our capabilities for design and communication, every advance in either separately requires a momentous effort. Each success, however, allows a corresponding effort on the other side to reach a little further. At some point the reaction will "go," and we will find ourselves at the favorable equilibrium point of the system, possessing mechanisms that are both highly intelligent and communicative.

With this view of the task and its setting, we can turn to the substance of the report: the development of chess programs. We will proceed historically, since this arrangement of the material will show most clearly what progress is being made in obtaining systems of increasing complexity and intelligence.

Shannon's Proposal

The relevant history begins with a paper by Claude Shannon in 1949. He did not present a particular chess program, but discussed many of the basic problems involved. The framework he introduced has guided most of the subsequent analysis of the problem.

As Shannon observed, chess is a finite game. There is only a finite number of positions, each of which admits a finite number of alternative moves. The rules of chess assure that any play will terminate: that eventually a position will be reached that is a win, loss, or draw. Thus chess can be completely described as a branching tree (as in Fig. 2), the nodes corresponding to positions and the branches corresponding to the alternative moves from each position. It is intuitively clear, and easily proved, that for a player who can view the entire tree and see all the ultimate consequences of each alternative, chess becomes a simple game. Starting with the terminal positions, which have determinate payoffs, he can work backwards, determining at each node which branch is best for him or his opponent as the case may be, until he arrives at the alternative for his next move.

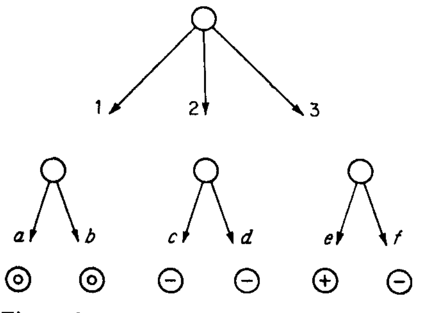


Figure 2. The game tree and minimaxing.

This inferential procedure—called *minimaxing* in the theory of games—is basic to all the attempts so far to program computers for chess. Let us be sure we understand it. Figure 2 shows a situation where White is to move and has three choices, (1), (2), and (3). White's move will be followed by Black's: (*a*) or (*b*) in case move (1) is made; (*c*) or (*d*) if move (2) is made; and (*e*) or (*f*) if move (3) is made. To keep the example simple, we have assumed that all of Black's moves lead to positions with known payoffs: (+) meaning a win for White, (0) meaning a draw, and (−) meaning a loss for White. How should White decide what to do—what inference procedure allows him to determine which of the three moves is to be preferred? Clearly, no matter what Black does, move (1) leads to a draw. Similarly, no matter what Black does, move (2) leads to a loss for White. White should clearly prefer move (1) to move (2). But what about move (3)? It offers the possibility of a win, but also contains the possibility of a loss; and furthermore, the outcome is in Black's control. If White is willing to impute any analytic ability to his opponent, he must conclude that move (3) will end as a loss for White, and hence that move (1) is the preferred move. The win from move (3) is completely insubstantial, since it can never be realized. Thus White can impute a value to a position—in this case draw—by reasoning backward from known values.

To repeat: If the entire tree can be scanned, the best move can be determined simply by the minimaxing procedure. Now minimaxing might have been the "wheel" of chess—with the adventure ended almost before it had started—if the tree were not so large that even current computers can discover only the minutest fraction of it in years of computing. Shannon's estimate, for instance, is that there are something like 10^{120} continuations to be explored, with less than 10^{16} microseconds available in a century to explore them.

Shannon then suggested the following framework. Playing chess consists of considering the alternative moves, obtaining some effective evaluation of them by means of analysis, and choosing the preferred alternative on the basis of the evaluation. The analysis—which is the hard part—could be factored into three parts. First, one would explore the continuations to a certain depth. Second, since it is clear that the explorations cannot be deep enough to reach terminal positions, one would evaluate the positions reached at the end of each exploration in terms of the pattern of men on the chessboard. These static evaluations would then be combined by means of the minimaxing procedure to form the effective value of the alternative. One would then choose the move with the highest effective value. The rationale behind this factorization was the reasonableness that, for a given evaluation function, the greater the depth of analysis, the better the chess that would be played. In the limit, of course, such a process would play perfect chess by finding terminal positions for all continuations. Thus a metric was provided that measured all programs along the single dimension of their depth of analysis.

To complete the scheme, a procedure was needed to evaluate positions statically—that is, without making further moves. Shannon proposed a numerical measure formed by summing, with weights, a number of factors or scores that could be computed for any position. These scores would correspond to the various features that chess experts assert are important. This approach gains plausibility from the existence of a few natural quantities in chess, such as the values of pieces, and the mobility of men. It also gains plausibility, of course, from the general use in science and engineering of linearizing assumptions as first approximations.

To summarize: the basic framework introduced by Shannon for thinking about chess programs consists of a series of questions:

1. Alternatives
   Which alternative moves are to be considered?
2. Analysis
   a. Which continuations are to be explored and to what depth?
   b. How are positions to be evaluated statically—in terms of their patterns?
   c. How are the static evaluations to be integrated into a single value for an alternative?
3. Final choice procedure
   What procedure is to be used to select the final preferred move?

We would hazard that Shannon's paper is chiefly remembered for the specific answers he proposed to these questions: consider all alternatives; search all continuations to fixed depth, n; evaluate with a numerical sum; minimax to get the effective value for an alternative; and then pick the best one. His article goes beyond these specifics, however, and discusses the possibility of selecting only a small number of alternatives and continuations. It also discusses the possibility of analysis in terms of the functions that chessmen perform—blocking, attacking, defending. At this stage, however, it was possible to think of chess programs only in terms of extremely systematic procedures. Shannon's specific proposals have gradually been realized in actual programs, whereas the rest of his discussion has been largely ignored. And when proposals for more complex computations enter the research picture again, it is through a different route.

Turing's Program

Shannon did not present a particular program. His specifications still require large amounts of computing for even such modest depths of analysis as two or three moves. It remained for A. M. Turing (1950) to describe a program along these lines that was sufficiently simple to be simulated by hand, without the aid of a digital computer.

In Table 1 we have characterized Turing's program in terms of the framework just defined. There are some additional categories which will become clear as we proceed. The table also provides similar information for each of the other three programs we will consider.

TABLE 1 Comparison of Current Chess Programs

	Turing	Kister, Stein, Ulam, Walden, Wells (Los Alamos)	Bernstein, Roberts, Arbuckle, Belsky (Bernstein)	Newell, Shaw, Simon (NSS)
Vital statistics				
Date	1951	1956	1957	1958
Board	8 × 8	6 × 6	8 × 8	8 × 8
Computer	Hand simulation	MANIAC-I 11,000 ops./sec	IBM 704 42,000 ops./sec	RAND JOHNNIAC 20,000 ops./sec
Chess program				
Alternatives	All moves	All moves	7 plausible moves Sequence of move generators	Variable Sequence of move generators
Depth of analysis	Until dead (exchanges only)	All moves 2 moves deep	7 plausible moves 2 moves deep	Until dead Each goal generates moves
Static evaluation	Numerical Many factors	Numerical Material, mobility	Numerical Material, mobility Area control King defense	Nonnumerical Vector of values Acceptance by goals
Integration of values	Minimax	Minimax (modified)	Minimax	Minimax
Final choice	Material dominates Otherwise, best value	Best value	Best value	1. First acceptable 2. Double function
Programming				
Language		Machine code	Machine code	IPL-IV, interpretive
Data scheme		Single board No records	Single board Centralized tables Recompute	Single board Decentralized List structure Recompute
Time	Minutes	12 min/move	8 min/move	1–10 hr/move (est.)
Space		600 words	7000 words	Now 6000 words, est. 16,000
Results				
Experience	1 game	3 games (no longer exists)	2 games	0 games Some hand simulation
Description	Loses to weak player Aimless Subtleties of evaluation lost	Beats weak player Equivalent to human with 20 games experience	Passable amateur Blind spots Positional	Good in spots (opening) No aggressive goals yet

Turing's program considered all alternatives—that is, all legal moves. In order to limit computation, however, he was very circumspect about the continuations the program considered. Turing introduced the notion of a "dead" position: one that in some sense was stable, hence could be evaluated. For example, there is no sense in counting material on the board in the middle of an exchange of Queens: one should explore the continuations until the exchange has been carried through—to the point where the material is not going to change with the next move. So Turing's program evaluated material at dead positions only. He made the value of material dominant in his static evaluation, so that a decision problem remained only if minimaxing revealed several alternatives that were equal in material. In these cases, he applied a supplementary additive evaluation to the positions reached by making the alternative moves. This evaluation included a large number of factors—mobility, backward pawns, defense of men, and so on—points being assigned for each.

Thus Turing's program is a good instance of a chess-playing system as envisaged by Shannon, although a small-scale one in terms of computational requirements. Only one published game, as far as we know, was played with the program. It proved to be rather weak, for it lost against a weak human player (who did not know the program, by the way), although it was not entirely a pushover. In general its play was rather aimless, and it was capable of gross blunders, one of which cost it the game. As one might have expected, the subtleties of the evaluation function were lost upon it. Most of the numerous factors included in the function rarely had any influence on the move chosen. In summary: Turing's program was not a very good chess player, but it reached the bottom rung of the human ladder.

There is no *a priori* objection to hand simulation of a program, although experience has shown that it is almost always inexact for programs of this complexity. For example, there is an error in Turing's play of his program, because he—the human simulator—was unwilling to consider all the alternatives. He failed to explore the ones he "knew" would be eliminated anyway, and was wrong once. The main objection to hand simulation is the amount of effort required to do it. The computer is really the enabling condition for exploring the behavior of a complex program. One cannot even realize the potentialities of the Shannon scheme without programming it for a computer.

The Los Alamos Program

In 1956 a group at Los Alamos programmed MANIAC I to play chess (Kister et al., 1957).[1] The Los Alamos program is an almost perfect

[1] There are two other explorations between 1951 and 1956 of which we are aware

example of the type of system specified by Shannon. As shown in the table, all alternatives were considered; all continuations were explored to a depth of two moves (*i.e.,* two moves for Black and two for White); the static evaluation function consisted of a sum of material and mobility measures; the values were integrated by a minimax procedure,[2] and the best alternative in terms of the effective value was chosen for the move.

In order to carry out the computation within reasonable time limits, a major concession was required. Instead of the normal chessboard of eight squares by eight squares, they used a reduced board, six squares by six squares. They eliminated the Bishops and all special chess moves: castling, two-square Pawn moves in the opening, and *en passant* captures.

The result? Again the program is a weak player, but now one that is capable of beating a weak human player, as the machine demonstrated in one of its three games. It is capable of serious blunders, a common characteristic, also, of weak human play.

Since this is our first example of actual play on a computer, it is worth looking a bit at the programming and machine problems. In a normal 8×8 game of chess there are about 30 legal alternatives at each move, on the average, thus looking two moves ahead brings 30^4 continuations, about 800,000, into consideration. In the reduced 6×6 game, the designers estimate the average number of alternatives at about 20, giving a total of about 160,000 continuations per move. Even with this reduction of five to one, there are still a lot of positions to be looked at. By comparison, the best evidence suggests that a human player considers considerably less than 100 positions in the analysis of a move (De Groot, 1946). The Los Alamos program was able to make a move in about 12 minutes on the average. To do this the code had to be very simple and straightforward. This can be seen by the size of the program—only 600 words. In a sense, the machine barely glanced at each position it evaluated. The two measures in the evaluation function are obtained directly from the process of looking at continuations: changes in material are noticed if the moves are captures, and the mobility score for a position is equal to the number of new positions to which it leads—hence is computed almost without effort when exploring all continuations.

The Los Alamos program tests the limits of simplification in the direction of minimizing the amount of information required for each position evaluated, just as Turing's program tests the limits in the direction of minimizing the amount of exploration of continuations. These programs, espe-

—a hand simulation by F. Mosteller and a Russian program for BESM. Unfortunately, not enough information is available on either to talk about them, so we must leave a gap in the history between 1951 and 1956.

[2] The minimax procedure was a slight modification of the one described earlier, in that the mobility score for each of the intermediate positions was added in.

cially the Los Alamos one, provide real anchor points. They show that, with very little in the way of complexity, we have at least entered the arena of human play—we can beat a beginner.

Bernstein's Program

Over the last two years Alex Bernstein, a chess player and programmer at IBM, has constructed a chess-playing program for the IBM 704 (for the full 8 × 8 board) (Bernstein and Roberts, 1958*b*; Bernstein et al., 1958*a*). This program has been in partial operation for the last six months, and has now played one full game plus a number of shorter sequences. It, too, is in the Shannon tradition, but it takes an extremely important step in the direction of greater sophistication: only a fraction of the legal alternatives and continuations are considered. There is a series of subroutines, which we can call plausible move generators, that propose the moves to be considered. Each of these generators is related to some feature of the game: King safety, development, defending own men, attacking opponent's men, and so on. The program considers at most seven alternatives, which are obtained by operating the generators in priority order, the most important first, until the seven are accumulated.

The program explores continuations two moves ahead, just as the Los Alamos program did. However, it uses the plausible move generators at each stage, so that, at most, 7 direct continuations are considered from any given position. For its evaluation function it uses the ratio of two sums, one for White and one for Black. Each sum consists of four weighted factors: material, King defense, area control, and mobility. The program minimaxes and chooses the alternative with the greatest effective value.

The program's play is uneven. Blind spots occur that are very striking; on the other hand it sometimes plays very well for a series of moves. It has never beaten anyone, as far as we know; in the one full game it played it was beaten by a good player, (Bernstein and Roberts, 1958*b*), and it has never been pitted against weak players to establish how good it is.

Bernstein's program gives us our first information about radical selectivity, in move generation and analysis. At 7 moves per position, it examines only 2,500 final positions two moves deep, out of about 800,000 legal continuations. That it still plays at all tolerably with a reduction in search by a factor of 300 implies that the selection mechanism is fairly effective. Of course, the selections follow the common and tested lore of the chess world; so that the significance of the reduction lies in showing that this lore is being successfully captured in mechanism. On the other hand, such radical selection should give the program a strong proclivity to overlook moves and consequences. The selective mechanisms in Bernstein's program have none of the checks and balances that exist in human selection on the chessboard. And this is what we find. For example, in one situation a Bishop was successively attacked by three Pawns, each time retreating one square to a post where the next Pawn could attack it. The program remained oblivious to this possibility since the successive Pawn pushes that attacked the Bishop were never proposed as plausible moves by the generators. But this is nothing to be unhappy about. Any particular difficulty is removable: in the case of the Bishop, by adding another move generator responsive to another feature of the board. This kind of error correction is precisely how the body of practical knowledge about chess programs and chess play will accumulate, gradually teaching us the right kinds of selectivity.

Every increase in sophistication of performance is paid for by an increase in the complexity of the program. The move generators and the components of the static evaluation require varied and diverse information about each position. This implies both more program and more computing time per position than with the Los Alamos program. From Table 1, we observe that Bernstein's program takes 7000 words, the Los Alamos program only 600 words: a factor of about 10. As for time per position, both programs take about the same time to produce a move—8 and 12 minutes respectively. Since the increase in problem size of the 8×8 board over the 6×6 board (about 5 to 1) is approximately canceled by the increase in speed of the IBM 704 over the MANIAC (also about 5 to 1, counting the increased power of the 704 order code), we can say they would both produce moves in the same 8×8 game in the same time. Hence the increase in amount of processing per move in Bernstein's program approximately cancels the gain of 300 to 1 in selectivity that this more complex processing achieves. This is so, even though Bernstein's program is coded to attain maximum speed by the use of fixed tables, direct machine coding, and so on.

We have introduced the comparison in order to focus on computing speed versus selectivity as sources of improvement in complex programs. It is not possible, unfortunately, to compare the two programs in performance level except very crudely. We should compare an 8×8 version of the Los Alamos program with the Bernstein program, and we also need more games with each to provide reliable estimates of performance. Since the 8×8 version of the Los Alamos program will be better than the 6×6, compared to human play, let us assume for purposes of argument that the Los Alamos and Bernstein programs are roughly comparable in performance. To a rough approximation, then, we have two programs that achieve the same quality of performance with the same total effort by two different routes: the Los Alamos program by using no selectivity and being very fast, and the Bernstein program by using a large amount of selectivity and taking much more effort per position examined in order to make the selection.

The point we wish to make is that this equality is an accident: that selectivity is a very powerful device and speed a very weak device for improving the performance of complex programs. For instance, suppose both the Los Alamos and the Bernstein programs were to explore three moves deep instead of two as they now do. Then the Los Alamos program would take about 1000 times (30^2) as long as now to make a move, whereas Bernstein's program would take about 50 times as long (7^2), the latter gaining a factor of 20 in the total computing effort required per move. The significant feature of chess is the exponential growth of positions to be considered with depth of analysis. As analysis deepens, greater computing effort per position soon pays for itself, since it slows the growth in number of positions to be considered. The comparison of the two programs at a greater depth is relevant since the natural mode of improvement of the Los Alamos program is to increase the speed enough to allow explorations three moves deep. Furthermore, attempts to introduce selectivity in the Los Alamos program will be extremely costly relative to the cost of additional selectivity in the Bernstein program.

One more calculation might be useful to emphasize the value of heuristics that eliminate branches to be explored. Suppose we had a branching tree in which our program was exploring n moves deep, and let this tree have four branches at each node. If we could double the speed of the program—that is, consider twice as many positions for the same total effort—then this improvement would let us look half a move deeper ($n + \frac{1}{2}$). If, on the other hand, we could double the selectivity—that is, only consider two of the four branches at each node, then we could look twice as deep ($2n$). It is clear that we could afford to pay an apparently high computing cost per position to achieve this selectivity.

To summarize, Bernstein's program introduces both sophistication and complication to the chess program. Although in some respects—*e.g.,* depth of analysis—it still uses simple uniform rules, in selecting moves to be considered it introduces a set of powerful heuristics which are taken from successful chess practice, and drastically reduce the number of moves considered at each position.

Newell, Shaw, and Simon Program

Although our own work on chess started in 1955, it took a prolonged vacation during a period in which we were developing programs that discover proofs for theorems in symbolic logic (Newell, Shaw, and Simon, 1957; Newell and Simon, 1956). In a fundamental sense, proving theorems and playing chess involve the same problem: reasoning with heuristics that select fruitful paths of exploration in a space of possibilities that grows exponentially. The same dilemmas of speed versus selection and uniformity versus sophistication exist in both problem domains. Likewise, the programming costs attendant upon complexity seem similar for both. So we have recently returned to the chess programming problem equipped with ideas derived from the work on logic.

The historical antecedents of our own work are somewhat different from those of the other investigators we have mentioned. We have been primarily concerned with describing and understanding human thinking and decision processes (Newell, Shaw, and Simon, 1958*a*, 1958*c*). However, both for chess players and for chess programmers, the structure of the task dictates in considerable part the approach taken, and our current program can be described in the same terms we have used for the others. Most of the positive features of the earlier programs are clearly discernible: The basic factorization introduced by Shannon; Turing's concept of a dead position; and the move generators, associated with features of the chess situation, used by Bernstein. Perhaps the only common characteristic of the other programs that is strikingly absent from ours—and from human thinking also, we believe—is the use of numerical additive evaluation functions to compare alternatives.

Basic Organization

Figure 3 shows the two-way classification in terms of which the program is organized. There is a set of goals, each of which corresponds to some feature of the chess situation—King safety, material balance, center control, and so on. Each goal has associated with it a collection of processes, corresponding to the categories outlined by Shannon: a move generator, a static evaluation routine, and a move generator for analysis. The routine for integrating the static evaluations into an effective value for a proposed move, and the final choice procedure are both common routines for the whole program, and therefore are not present in each separate component.

Goals

The goals form a basic set of modules out of which the program is constructed. The goals are independent: any of them can be added to the

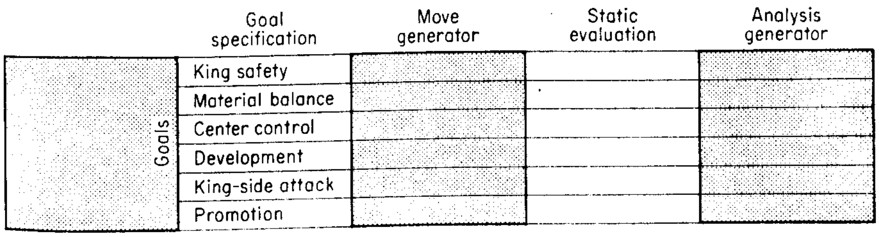


Figure 3. Basic organization of the NSS chess program.

program or removed without affecting the feasibility of the remaining goals. At the beginning of each move a preliminary analysis establishes that a given chess situation (a "state") obtains, and this chess situation evokes a set of goals appropriate to it. The goal specification routines shown for each goal in Fig. 3 provide information that is used in this initial selection of goals. The goals are put on a list with the most crucial ones first. This goal list then controls the remainder of the processing: the selection of alternatives, the continuations to be explored, the static evaluation, and the final choice procedure.

What kind of game the program will play clearly depends on what goals are available to it and chosen by it for any particular move. One purpose of this modular construction is to provide flexibility over the course of the game in the kinds of considerations the program spends its effort upon. For example, the goal of denying stalemate to the opponent is relevant only in certain end-game situations where the opponent is on the defensive and the King is in a constrained position. Another purpose of the modular construction is to give us a flexible tool for investigating chess programs—so that entirely new considerations can be added to an already complex but operational program.

Move Generation

The move generator associated with each goal proposes alternative moves relevant to that goal. These move generators carry the burden of finding positive reasons for doing things. Thus, only the center-control generator will propose P-Q4 as a good move in the opening; only the material-balance generator will propose moving out of danger a piece that is *en prise*. These move generators correspond to the move generators in Bernstein's program, except that here they are used exclusively to generate alternative moves and are not used to generate the continuations that are explored in the course of analyzing a move. In Bernstein's program—and *a fortiori* in the Los Alamos program—identical generators are used both to find a set of alternative moves from which the final choice of next move is made, and also to find the continuations that must be explored to assess the consequences of reaching a given position. In our program the latter function is performed by a separate set of analysis generators.

Evaluation

Each move proposed by a move generator is assigned a value by an analysis procedure. We said above that the move generators have the responsibility for finding positive reasons for making moves. Correspondingly, the analysis procedure is concerned only with the acceptability of a move once it has been generated. A generator proposes; the analysis procedure disposes.

The value assigned to a move is obtained from a series of evaluations, one for each goal. The value is a vector, if you like to think of it that way, except that it does not necessarily have the same components throughout the chess game, since the components derive from the basic list of goals that is constructed from the position at the beginning of each move. Each component expresses acceptability or unacceptability of a position from the viewpoint of the goal corresponding to that component. Thus, the material-balance goal would assess only the loss or gain of material; the development goal, the relative gain or loss of *tempi;* the Pawn structure goal, the doubling and isolation of Pawns; and so on. The value for a component is in some cases a number—*e.g.,* in the material-balance goal where we use conventional piece values: 9 for a Queen, 5 for a Rook, and so on. In other cases the component value is dichotomous, simply designating the presence or absence of some property, like the blocking of a move or the doubling of a Pawn.

As in the other chess programs, our analysis procedure consists of three parts: exploring continuations to some depth, forming static evaluations, and integrating these to establish an effective value for the move. By a process that we will describe later, the analysis move generators associated with the goals determine what branches will be explored from each position reached. At the final position of each continuation, a value is assigned using the static evaluation routines of each goal to provide the component values. The effective value for a proposed move is obtained by minimaxing on these final static values. Minimaxing seems especially appropriate for an analysis procedure that is inherently conservative, such as an acceptance test.

To be able to minimax, it must be possible to compare any two values and decide which is preferable, or whether they are equal in value. For values of the kind we are using, there must be a complete ordering on the vectors that determine them. Further, this ordering must allow variation in the size and composition of the goal list. We use a lexicographic ordering: Each component value is completely ordered within itself; and higher priority values completely dominate lower priority values, as determined by the order of goals on the goal list. To compare two values, then, the first components are compared. If one of these is preferable to the other, this determines the preference for the entire value. If the two components are equal, then the second pair of components is compared. If these are unequal in value, they determine the preference for the entire value; otherwise the next components are compared, and so on.

Final Choice

It is still necessary to select the move to be played from the alternative moves, given the values assigned to them by the analysis procedure. In the other programs the final choice procedure was simply an extension of the minimax: choose the one with highest value. Its obviousness rests on the assumption that the set of alternatives to be considered is a fixed set. If this assumption is relaxed, by generating alternatives sequentially, then other procedures are possible. The simplest, and the one we are currently using, is to set an acceptance level as final criterion and simply take the first acceptable move. The executive routine proceeds down the goal list, activating the move generators of the goals in order of priority, so that important moves are considered first. The executive saves the best move that has been found up to any given moment, and if no moves reach the specified level of acceptability, it makes the best move that was found.

Another possible final choice procedure is to search for an acceptable move that has a double function—that is, a move that is proposed by more than one generator as having a positive effect. With this plan, the executive proceeds down the list of goals in order of priority. After finding an acceptable move, it activates the rest of the generators to see if the move will be proposed a second time. If not, it works from the list of unevaluated moves just obtained to see if any move proposed twice is acceptable. If not, it takes the first acceptable move or the best if none has proved acceptable. This type of executive has considerable plausibility, since the concept of multiple function plays an important role in the chess literature.

Yet a third variation in the final choice procedure is to divide the goals into two lists. The first list contains all the features that should normally be attended to; the second list contains features that are rare in occurrence but either very good or very bad if they do occur. On this second list would be goals that relate to sacrificial combinations, hidden forks or pins that are two moves away, and so on. The executive finds an acceptable move with the first, normal list. Then the rest of the available time is spent looking for various rare consequences derived from the second list.

Analysis

In describing the basic organization of the program we skipped over the detailed mechanism for exploring continuations, simply assuming that certain continuations were explored, the static values computed, and the effective value obtained by minimaxing. But it is clear that the exact mechanisms are very important. The analysis move generators are the main agents of selectivity in the program: They determine for each position arrived at in the analysis just which further branches must be explored, hence the average number of branches in the exploration tree and its average depth. The move generators for the alternatives and the final choice procedure also affect the amount of exploration by determining what moves are considered. But their selection operates only once per move, whereas the selectivity of the analysis generators operates at each step (half move) of the exploration. Hence the selectivity of the analysis generators varies geometrically with the average depth of analysis.

The exploration of continuations is based on a generalization of Turing's concept of a dead position. Recall that Turing applied this notion to exchanges, arguing that it made no sense to count material on the board until all exchanges that were to take place had been carried out. We apply the same notion to each feature of the board: The static evaluation of a goal is meaningful only if the position being evaluated is "dead" with respect to the feature associated with that goal—that is, only if no moves are likely to be made that could radically alter that component static value. The analysis move generators for each goal determine for any position they are applied to whether the position is dead with respect to their goal; if not, they generate the moves that are both plausible and might seriously affect the static value of the goal. Thus the selection of continuations to be explored is dictated by the search for a position that is dead with respect to all the goals, so that, finally, a static evaluation can be made. Both the number of branches from each position and the depth of the exploration are controlled in this way. Placid situations will produce search trees containing only a handful of positions; complicated middle game situations will produce much larger ones.

To make the mechanics of the analysis clearer, Fig. 4 gives a schematic example of a situation. P_0 is the initial position from which White, the machine, must make a move. The arrow, α, leading to P_1 represents an alternative proposed by some move generator. The move is made internally (*i.e.*, "considered"), yielding position P_1, and the analysis procedure must then obtain the value of P_1, which will become the value imputed to the proposed alternative, α. Taking each goal from the goal list in turn, an attempt is made to produce a static evaluation. For P_1 this attempt is successful for the first and second components, yielding values of 5 and 3 respectively. (Numbers are used for values throughout this example to keep the picture simple; in reality, various sets of ordered symbols are used, their exact structure depending on the nature of the computation.) However, the third component does not find the position dead, and generates two moves, β and γ. The first, β, is considered, leading to P_2, and an attempt is made to produce a static evaluation of it. This proceeds just as with P_1, except that this time all components find the

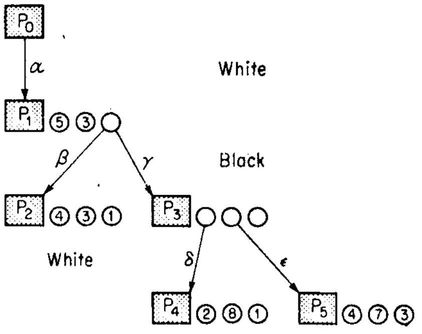


Figure 4. Analysi

position dead and the static value (4,3,1) is obtained. Then the second move, γ, from P_1 is considered, leading to P_3. The attempt to produce a static value for P_3 runs into difficulties with the first component, which generates one move, δ, to resolve the instability of P_3 with respect to its feature. This move leads to P_4 which is evaluable, having the value (2,8,1). However, the second component also finds P_3 not dead and generates a single move, ϵ, leading to P_5. This is also evaluable, having the value (4,7,3). The third component finds P_3 dead and therefore contributes no additional moves. Thus the exploration comes to an end with all terminal positions yielding complete static values. Since it is White's move at P_3, White will choose the move with the highest value. This is ϵ, the move to P_5, with a value of (4,7,3) (the first component dominates). The value of this move is the effective value assigned to P_3. Black now has a choice between the move, β, to P_2, yielding (4,3,1) and the move, γ, to P_3, yielding (4,7,3). Since Black is minimizing, he will choose β. This yields (4,3,1) as the effective value of the alternative, α, that leads to P_1, and the end of the analysis.

The minimaxing operation is conducted concurrently with the generation of branches. Thus if P_5, which has a value of (4,7,3), had been generated prior to P_4 no further moves would have been generated from P_3, since it is already apparent that Black will prefer P_2 to P_3. The value of P_3 is at least as great as the value of P_5, since it is White's move and he will maximize.

This analysis procedure is not a simple one, either conceptually or technically. There are a number of possible ways to terminate search and reach an effective evaluation. There is no built-in rule that guarantees that the search will converge; the success depends heavily on the ability to evaluate statically. The more numerous the situations that can be recognized as having a certain value without having to generate continuations, the more rapidly the search will terminate. The number of plausible moves that affect the value is also of consequence, as we discussed in connection with Bernstein's program, but there are limits beyond which this cannot be reduced. For example, suppose that a position is not dead with respect to Material Balance and that one of the machine's pieces is attacked. Then it can try to (*a*) take the attacker, (*b*) add a defender, (*c*) move the attacked piece, (*d*) pin the defender, (*e*) interpose a man between the attacker and the attacked, or (*f*) launch a counterattack. Alternatives of each of these types must be sought and tried—they are all plausible and may radically affect the material balance.

As an example of the heuristics involved in achieving a static evaluation, imagine that the above situation occurred after several moves of an exploration, and that the machine was already a Pawn down from the early part of the continuation. Then, being on the defensive implies a very remote chance of recovering the Pawn. Consequently, a negative value of at least a Pawn can be assigned to the position statically. This is usually enough in connection with concurrent minimaxing to eliminate the continuation from further consideration.

Summary

Let us summarize our entire program. It is organized in terms of a set of goals: these are conceptual units of chess—King safety, passed Pawns, and so on. Each goal has several routines associated with it:

1. A routine that specifies the goal in terms of the given position;
2. A move generator that finds moves positively related to carrying out the goal;
3. A procedure for making a static evaluation of any position with respect to the goal, which essentially measures acceptability;
4. An analysis move generator that finds the continuations required to resolve a situation into dead positions.

The alternative moves come from the move generators, considered in the order of priority of their respective goals. Each move, when it is generated, is subjected to an analysis. This analysis generates an exploration of the continuations following from the move until dead positions are reached and static evaluations computed for them. The static evaluations are compared, using minimax as an inference procedure, so that an effective value is eventually produced for each alternative. The final choice procedure can rest on any of several criteria: for instance, choosing the first move generated that has an effective value greater than a given norm.

Examples of Goals

In this section we will give two examples of goals and their various components to illustrate the type of program we are constructing. The first example is the center-control goal:

CENTER CONTROL

Specification. Goal is always operative unless there are no more center Pawns to be moved to the fourth rank.

Move Generator

1. Move P-Q4, P-K4 (primary moves).
2. Prevent the opponent from making his primary moves.
3. Prepare your own primary moves:
   a. Add a defender to Q4 or K4 square.
   b. Eliminate a block to moving QP or KP.

Static Evaluation. Count the number of blocks to making the primary moves.
Analysis Move Generators. None; static evaluation is always possible.

To interpret this a little: Goals are proposed in terms of the general situation—*e.g.,* for the opening game. The list of goals is made up for a position by applying, in turn, the specification of each of the potential goals. Whether any particular goal is declared relevant or irrelevant to the position depends on whether or not the position meets its specification. For Center Control, no special information need be gathered, but the goal is declared irrelevant if the center Pawns have already been moved to the fourth rank or beyond.

The most important part of the center-control program is its move generator. The generator is concerned with two primary moves: P-Q4 and P-K4. It will propose these moves, if they are legal, and it is the responsibility of the analysis procedures (for all the goals) to reject the moves if there is anything wrong with them—*e.g.,* if the Pawns will be taken when moved. So, after 1. P-Q4, P-Q4, the center-control move generator will propose 2. P-K4, but (as we shall see) the evaluation routine of the material balance goal will reject this move because of the loss of material that would result from 2. . . . , P $\times$ P. The center-control generator will have nothing to do with tracing out these consequences.

If the primary moves cannot be made, the center-control move generator has two choices: to prepare them, or to prevent the opponent from making his primary moves. The program's style of play will depend very much on whether prevention has priority over preparation (as it does in our description of the generator above), or vice versa. The ordering we have proposed, which puts prevention first, probably produces more aggressive and slightly better opening play than the reverse ordering. Similarly, the style of play depends on whether the Queen's Pawn or the King's Pawn is considered first.

The move generator approaches the subgoal of preventing the opponent's primary moves (whenever this subgoal is evoked) in the following way. It first determines whether the opponent can make one of these moves by trying the move and then obtaining an evaluation of it from the opponent's viewpoint. If one or both of the primary moves are not rejected, preventive moves will serve some purpose. Under these conditions, the center-control move generator will generate them by finding moves that bring another attacker to bear on the opponent's K4 and Q4 squares or that pin a defender of one of these squares. Among the moves this generator will normally propose are N-B3 and BP-B4.

The move generator approaches the subgoal of preparing its own primary moves by first determining why the moves cannot be made without preparation—that is, whether the Pawn is blocked from moving by a friendly piece, or whether the fourth-rank square is unsafe for the Pawn. In the former case, the generator proposes moves for the blocking piece; in the latter case, it finds moves that will add defenders to the fourth-rank square, drive away or pin attackers, and so on.

So much for the center-control move generators. The task of the evaluation routine for the center-control goal is essentially negative—to assure that moves, proposed by some other goal, will not be made that jeopardize control of the center. The possibility is simply ignored that a move generator for some other goal will inadvertently provide a move that contributes to center control. Hence, the static evaluation for Center Control is only concerned that moves not be made that interfere with P-K4 and P-Q4. A typical example of a move that the center-control evaluation routine is prepared to reject is B-Q3 or B-K3 before the respective center Pawns have been moved.

The second example of a goal is Material Balance. This is a much more extensive and complicated goal than Center Control, and handles all questions about gain and loss of material in the immediate situation. It does not consider threats like pins and forks, where the actual exchange is still a move away; other goals must take care of these. Both the negative and positive aspects of material must be included in a single goal, since they compensate directly for each other, and material must often be spent to gain material.

MATERIAL BALANCE

Specification. A list of exchanges on squares occupied by own men, and a list of exchanges on squares occupied by opponent's men. For each exchange square there is listed the target man, the list of attackers, and the list of defenders (including, *e.g.*, both Rooks if they are doubled on the appropriate rank or file). For each exchange square a static exchange value is computed by playing out the exchange with all the attackers and defenders assuming no indirect consequences like pins, discovered attacks, etc. Exchange squares are listed in order of static exchange value, largest negative value first. Squares with positive values for the defender are dropped from the list. At the same time a list of all pinned men is generated.

Move Generator. Starting with the exchange squares at the top of the list, appropriate moves are generated. If the most important exchange square is occupied by the opponent, captures by attacking pieces are proposed, the least valuable attacker being tried first. If the move is rejected because the attacker is pinned, the next attacker is tried. If the move is rejected for another reason, the possibility of exchange on this square is abandoned, and the next exchange square examined.

If the exchange square under examination is occupied by the program's own piece, a whole series of possible moves is generated:

a. Try "no move" to see if attack is damaging.
b. Capture the attacker.
c. Add a defender not employed in another defense.
d. Move the attacked piece.
e. Interpose a man between the attacker and the target; but not a man employed elsewhere, and not if the interposer will be captured.
f. Pin the attacker with a man not employed elsewhere and not capturable by the attacker.

Static Evaluation. For each exchange square, add the values of own men and subtract the values of opponent's men. Use conventional values: Q-9, R-5, B-N-3, P-1.

Move Generators toward Dead Positions. A position is dead for this goal only if there are no exchanges—that is, if the specification list defined above is empty. Then a static evaluation can be made. Otherwise, the various kinds of moves defined under the move generator are made to resolve the exchanges. However, various additional qualifications are introduced to reduce the number of continuations examined. For example, if in a particular exchange material has already been lost and a man is still under attack, the position is treated as dead, since it is unlikely that the loss will be recovered. When a dead position is reached, the static evaluation is used to find a value for the position.

It is impossible to provide here more than a sketchy picture of the heuristics contained in this one goal. It should be obvious from this brief description that there are a lot of them, and that they incorporate a number of implicit assumptions about what is important, and what isn't, on the chessboard.

Performance of the Program

We cannot say very much about the behavior of the program. It was coded this spring and is not yet fully debugged. Only two goals have been coded: Material Balance and Center Control. Development is fully defined as well as a Pawn structure goal sufficient for the opening, where its role is primarily to prevent undesirable structures like doubled Pawns. These four goals—Material Balance, Center Control, Development, and Pawn Structure—in this order seem an appropriate set for the first phase of the opening game. Several others—King Safety, Serious Threats, and Gambits—need to be added for full opening play. The serious threats goal could be limited initially to forks and pins.

We have done considerable hand simulation with the program in typical positions. Two examples will show how the goals interact. In Fig. 5 the

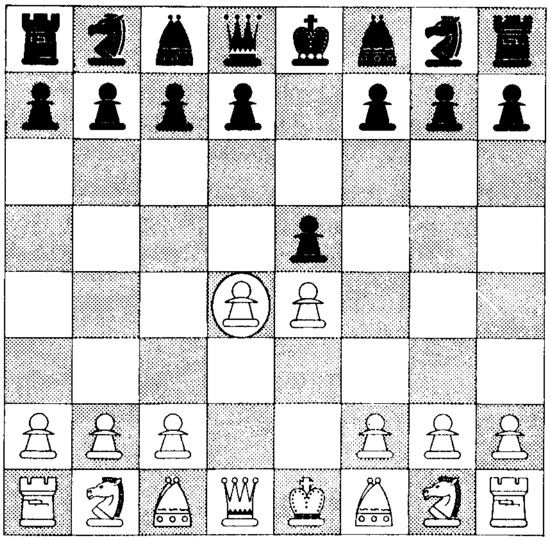


Figure 5.

machine is White and the play has been 1. P-K4, P-K4. Assuming the goal list mentioned above, the material-balance move generator will not propose any moves since there are no exchanges on the board. The center-control generator will propose P-Q4, which is the circled move in the figure. (In the illustration, we assume the center-control move generator has the order of the primary moves reversed from the order described earlier.) This move is rejected—as it should be—and it is instructive to see why. The move is proposed for analysis. Material Balance does not find the position dead, since there is an exchange, and generates Black's move, 2. . . . , P × P. The resulting position is still not dead, and 3. Q × P, is generated. The position is now dead for Material Balance, with no gain or loss in material. The first component of the static evaluation is "even." There are obviously no blocks to Pawn moves, so that the center control static value is acceptable. However, the third component, Development, finds the position not dead because there is now an exposed piece, the Queen. It generates replies that both attack the piece and develop—*i.e.,* add a tempo. The move 3. . . . , N-QB3 is generated. This forces a Queen move, resulting in loss of a tempo for White. Hence Development rejects the move, 2. P-Q4. (The move 3. . . . , B-B4 would not have sufficed for rejection by Development, since the Bishop could be taken.)

The second example, shown in Fig. 6, is from a famous game of Morphy against Duke Karl of Brunswick and Count Isouard. Play had proceeded 1. P-K4, P-K4; 2. N-KB3, P-Q3; 3. P-Q4. Suppose the machine is Black

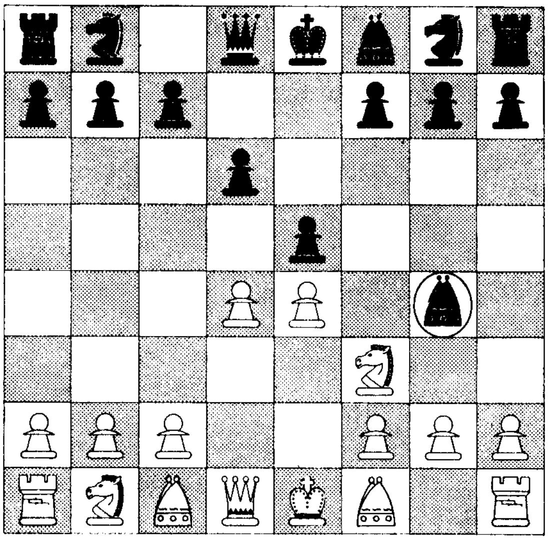


Figure 6.

in this position. The move 3. . . . , B-N5 is proposed by Material Balance to deal with the exchange that threatens Black with the loss of a Pawn. This is the move made by the Duke and Count. The analysis proceeds by 4. P $\times$ P, P $\times$ P. This opens up a new exchange possibility with the Queens, which is tried: 5. Q $\times$ Q, K $\times$ Q; 6. N $\times$ P. Thus the Pawn is lost in this continuation. Hence, alternative moves are considered at Black's nearest option, which is move 4, since there are no alternative ways of recapturing the Queen at move 5. The capture of White's Knight is possible, so we get: 4′. . . . , B $\times$ N; 5′. P $\times$ B, P $\times$ P; 6′. Q $\times$ Q, K $\times$ Q. This position is rejected by Development since the forced King move loses Black his castling privilege, and this loss affects the tempo count. This is a sufficient reason to reject the move 3. . . . , B-N5, without even examining the stronger continuation, 5″. Q $\times$ B, that Morphy as White chose. In our program, 5. P $\times$ B is generated before 5. Q $\times$ B. Either reply shows that 3. . . . , B-N5 is unsound.

One purpose of these examples is to illustrate a heuristic for constructing chess programs that we incline to rather strongly. We wish not only to have the program make good moves, but to do so for the right reasons. The chess commentary above is not untypical of human analysis. It also represents rather closely the analysis made by the program. We think this is sound design philosophy in constructing complex programs. To take another example: the four-goal opening program will not make sacrifices, and conversely, will always accept gambits. The existing program is unable to balance material against positional advantage. The way to make the program take account of sacrifices is to introduce an additional goal having to do with them explicitly. The corresponding heuristic for a human chess player is: don't make sacrifices until you understand what a sacrifice is. Stated in still another way, part of the success of human play depends on the emergence of appropriate concepts. One major theme in chess history, for example, is the emergence of the concept of the center and the notion of what it means to control the center. One should not expect the equivalent of such a concept simply to emerge from computation based on quite different features of the position.

Programming

The program we have been describing is extremely complicated. Almost all elements of the original framework put forward by Shannon, which were handled initially by simply uniform rules, have been made variable, and dependent on rather complicated considerations. Many special and highly particular heuristics are used to select moves and decide on evaluations. The program can be expected to be much larger, more intricate, and to require much more processing per position considered than even the Bernstein program.

In the introduction to this paper we remarked on the close connection between complexity and communication. Processes as complex as the Los Alamos program are unthinkable without languages like current machine codes in which to specify them. The Bernstein program is already a very complicated program in machine code; it involved a great deal of coding effort and parts of it required very sophisticated coding techniques. Our own program is already beyond the reach of direct machine coding: it requires a more powerful language.

In connection with the work on theorem-proving programs we have been developing a series of languages, called information processing languages (IPL's) (Newell, 1961*e*). The current chess program is coded in one of them, IPL-IV. An information processing language is an interpretive pseudocode—that is, there exists a program in JOHNNIAC machine code that is capable of intepreting a program in IPL and executing it. When operating, JOHNNIAC contains both the machine code and the IPL code.

It is not possible to give in this report a description of IPL-IV or of the programming techniques involved in constructing the chess program. Basically IPL is designed to manipulate lists, and to allow extremely complicated structures of lists to be built up during the execution of a program without incurring intolerable problems of memory assignment and program planning. It allows unlimited hierarchies of subroutines to be easily defined, and permits recursive definition of routines. As it stands —that is, prior to coding a particular problem—it is independent of subject matter (although biased toward list manipulation in the same sense that algebraic compilers are biased toward numerical evaluation of algebraic expressions). To code chess, a complete "chess vocabulary" is built up from definitions in IPL. This vocabulary consists of a set of processes for expressing basic concepts in chess: tests of whether a man bears on another man, or whether two men are on the same diagonal; processes for finding the direction between two men, or the first man in a given direction from another; and processes that express iterations over all men of a given type, or over all squares of a given rank. There are about 100 terms in this basic process vocabulary. The final chess program, as we have been describing it in this paper, is largely coded in terms of the chess vocabulary. Thus there are four language "levels" in the chess program: JOHNNIAC machine code, general IPL, basic chess vocabulary, and finally the chess program itself.

We can now make a rough assessment of the size and complexity of this program in comparison with the other programs. The table indicates that the program now consists of 6000 words and will probably increase to 16,000. The upper bound is dictated by the size of the JOHNNIAC drum and the fact that JOHNNIAC has no tapes. In terms of the pyramiding structure described above, this program is already much larger than Bernstein's, although it is difficult to estimate the "expansion" factor involved in converting IPL to machine code. (For one thing, it is not clear how an "equivalent" machine-coded program would be organized.) However, only about 1000 words of our program are in machine code, and 3000 words are IPL programs, some of which are as many as ten definitional steps removed from machine code. Further, all 12,000 words on the drum will be IPL program: no additional data or machine code are planned.

The estimated time per move, as shown in Table 1, is from one to ten hours, although moves in very placid situations like the opening will take only a few minutes. Even taking into account the difference in speed between the 704 and JOHNNIAC, our program still appears to be at least ten times slower than Bernstein's. This gap reflects partly the mismatch between current computers and computers constructed to do efficiently the kind of information processing required in chess (Shaw et al., 1958). To use an interpretive code, such as IPL, is in essence to simulate an "IPL computer" with a current computer. A large price has to be paid in computing effort for this simulation over and above the computing effort for the chess program itself. However, this gap also reflects the difficulty of specifying complex processes; we have not been able to write these programs and attend closely to the efficiency issue at the same time.

On both counts we have felt it important to explore the kind of languages and programming techniques appropriate to the task of specifying complex programs, and to ignore for the time being the costs we were incurring.

Conclusion

We have now completed our survey of attempts to program computers to play chess. There is clearly evident in this succession of efforts a steady development toward the use of more and more complex programs and more and more selective heuristics; and toward the use of principles of play similar to those used by human players. Partly, this trend represents—at least in our case—a deliberate attempt to simulate human thought processes. In even larger part, however, it reflects the constraints that the task itself imposes upon any information processing system that undertakes to perform it. We believe that any information processing system—a human, a computer, or any other—that plays chess successfully will use heuristics generically similar to those used by humans.

We are not unmindful of the radical differences between men and machines at the level of componentry. Rather, we are arguing that for tasks that could not be performed at all without very great selectivity—and chess is certainly one of these—the main goal of the program must be to achieve this selection. The higher-level programs involved in accomplishing this will look very much the same whatever processes are going on at more microscopic levels. Nor are we saying that programs will not be adapted to the powerful features of the computing systems that are used—*e.g.*, the high speed and precision of current digital computers, which seems to favor exploring substantial numbers of continuations. However, none of the differences known to us—in speed, memory, and so on—affect the essential nature of the task: search in a space of exponentially growing possibilities. Hence the adaptations to the idiosyncrasies of particular computers will all be secondary in importance, although they will certainly exist and may be worthwhile.

The complexity of heuristic programs requires a more powerful language for communicating with the computer than the language of elementary machine instructions. We have seen that this necessity has already mothered the creation of new information processing languages. But even with these powerful interpretive languages, communication with the machine is difficult and cumbersome. The next step that must be taken is to write programs that will give computers a problem-solving ability in understanding and interpreting instructions that is commensurable with their problem-solving ability in playing chess and proving theorems.

The interpreter that will transform the machine into an adequate student for a human instructor will not be a passive, algorithmic translator—as even the most advanced interpreters and compilers are today—but an active, complex, heuristic problem-solving program. As our explorations of heuristic programs for chess playing and other tasks teach us how to build such an interpreter, they will at last enable us to make the transition from the low-level equilibrium at which man-machine communication now rests to the high-level equilibrium that is certainly attainable.

Appendix: Game Played by H. A. Simon and the NSS Chess Program

The following chess game was played by the NSS chess program, CP-I. Its opponent was Prof. H. A. Simon, playing black. CP-I played white. The program was run on JOHNNIAC, and the moves each took 2 to 50 min of processing. The program has three goals: material balance, center control, and development. The lack of goals corresponding to king safety, serious threats, or pawn promotion seriously unbalances the play and makes the program insensitive to certain aspects of the play. Games by machines evoke commentary even more abundantly than do regular chess games. The italicized comments are those of Edward Lasker, a well-known chess master who has been much interested in chess machines; the other notes are by G. W. Baylor and S. M. Strassen.

	CP-I	*H. A. Simon*
1	P-Q4	N-KB3
2	N-QB3	

White prepares to occupy the center with P-K4, but the text move has the disadvantage of blocking the Queen's Bishop's Pawn, which when advanced to Queen Bishop Four, controls Queen Five with a pawn.

2	. . .	P-Q4
3	Q-Q3?	

This move does prepare P-K4; however, (1) minor pieces should generally be developed before the Queen, (2) the Queen is now subject to early attack by Black's minor pieces, and (3) the text move decreases the mobility of the King's Bishop.
In your game 3. Q-Q3 shows that you need an order that minor pieces should be developed ahead of the queen unless other orders in the program lead to the decision that a queen move is preferable.

3	. . .	P-QN3

3 . . . , P-QN3 is difficult to evaluate. Probably best was (*a*) 3 . . . , P-B4; if then 4. P-K4, P × KP; 5. N × P, P × P; 6. N × Nch, NP × N with a fine pawn plus. Also, considerable was (*b*) 3 . . . , P-KN3 so as to develop the Queen's Bishop on King Bishop Four, thus exploiting the misplacement of the White Queen.

4	P-K4	

Thematic!

4 . . .	B-N2
5 P × P	

Best I think. If, for example, (*a*) 5. P-K5, N-K5 gives Black strong control of Queen Four and King Five with a devastating P-QB4 to follow shortly; while (*b*) 5. P-B3 leaves White no good squares on which to develop his King side pieces.

P × P shows that your definition or "development" must probably be amplified to give a higher rating to moves which do not increase the mobility of one of the opponent's pieces.

5 . . .	N × P
6 N-B3	

White can effectively gain control of the center (especially Queen Five) with 6. N × N, Q × N (6 . . . , B × N is no better); 7. P-QB4!, Q-Q2; 8. N-KB3, P-K3; 9. B-K3 preventing Black's P-QB4 for a while. If, of course, 8 . . . , P-QB4; 9. P-Q5, P-K3 will be met simply by 10. P × P, in any case, with a good position for White.

6 . . .	P-K3

For now if 7. N × N, P × N is best because then the effect of 8. P-B4 is negated simply by 8 . . . , P × P which frees the Bishop and isolates the White Queen Pawn.

7 B-K2	

"A developing move and hence cannot be bad."

7 . . .	B-K2
8 B-K3	

Not bad: 8 . . . , N × B; 9. P × N is certainly not to be feared for when White gets P-K4 in, he will have the superior game. 8. B-K3 also has the added advantage of restraining Black's Queen Bishop Pawn. A more constructive placement of the pieces, however, might be accomplished by 8. N × N, 9. O-O, 10. B-KB4, and 11. R-K1 with strong control of King Five. And if . . . , P-QB4, then White can play P-QB3 effectively.

8 . . .	O-O
9 O-O	N-Q2
10 KR-K1	

The two Rook moves are not really good. White does not yet (and never will!) have a constructive plan: he is simply developing pieces on the center files where they are not necessarily optimally placed. Generally first rank Rook moves consolidate concrete plans. Thus White should attempt either to continue with (*a*) 10. N × N and 11. P-QB4 after which his Rooks will probably best be placed on Queen One and Queen Bishop One, or (*b*) 10. N-K5, N × N; 11. P × N after which the Queen file requires foremost attention. 10. N-K5 also enhances the mobility of the White King's Bishop which has been sadly restricted due to the misplacement of the White Queen (*i.e.*, B-KB3 will then be in order).

10 . . . P-QB4

Finally!

11 QR-Q1 Q-B2

Although this move does prevent 12. N-K5, it is not good. For instance on 12. N-QN5, Q-N1 (to be consistent); 13. P-B4!, N-N5; 14. Q-N1 threatening 15. P-QR3 and 16. P-Q5 is good for White so that 14 . . . , P × P; 15. N/5 × QP is probably in order for Black but still gives White the edge. Therefore Black should have continued pressure on the Queen Bishop file with 11 . . . , R-B1 and not have allowed the opportunity to White of playing 12. N-QN5 and 13. P-B4. Even after 11 . . . , R-B1, however, White could continue well with 12. N-K5.

12 N × N

Missing the sharpest continuation, but the text is not bad; *e.g.,* 12 . . . , P × N; 13. P-B4, P × QP; 14. B × P, P × P; 15. Q × BP with at least equality for White.

12 . . . B × N?

This allows the now strong continuation 13. P-B4 after which 13 . . . , B-N2; 14. P-Q5, P × P; 15. P × P yields a strong passed pawn (an immediate threat of 16. P-Q6) as well as control of the board.

13 P-QR4?

A terrible move: just defends the Queen Rook Pawn whereas the multifunctional 13. P-B4 defends the Queen Rook Pawn and also attacks the center.

I am wondering why your "center control" orders did not suggest 13. P-QB4 rather than P-QR4. It would really have given the machine a very good game. 13. P-QR4 shows that an order—or a series of orders—is missing which would lead to the preparation of protection of pawns located in a file the opponent has opened for a Rook.

13 . . . QR-B1

14 Q-B3

After 14 . . . , P × P; 15. Q × Q, R × Q; 16. N × P, White can solidify his position with P-QB3, but even so 14. Q-B3 doesn't really contribute anything to the position. 14. P-B4 is still best.

14 . . .	B-KB3!

Capitalizing on White's shortsightedness! 14 . . . , N-KB3 is also good (heading for King Five).

15 B-QN5	

Clever: Black was threatening to win a pawn with 15 . . . , P × P; 16. Q × Q, R × Q; 17. N × P, B × N; 18. B × B and 18 . . . , R × P. After the text move, however, the Queen Knight must be defended. The alternative (other than a Rook move) 15. B-Q3 does not actually defend the Queen Bishop Pawn because of 15 . . . , B × N; 16. P × B, P × P; 17. B × P (17. Q × Q, R × Q and White cannot recapture the pawn), Q-N1!; 18. Q-N4, P-QR4; 19. Q-N5, B × B; 20. Q × N, KR-Q1 with a strong attack for Black.

15 . . .	B × N

Good. If 15 . . . , KR-Q1 first, then 16. B × N, R × B; 17. N-K5, P × P; 18. B × P holding on admirably well.

16 P × B	KR-Q1
17 B × N?	

White loses his last opportunity to defend his Queen Bishop Pawn. Some Queen move, for instance 17, Q-Q2, holds the pawn: 17. Q-Q2, P × P; 18. B × P, B × B; 19. Q × B, Q × BP; 20. B × N winning (20 . . . , R-B2; 21. Q-KB4!, P-KR3; 22. R-Q2!).

17 . . .	Q × B
18 P-N3	

As good as many and better than some: White must lose a pawn anyhow.

18 . . .	P × P
19 Q-Q2!	

Very good. White finds the only way (other than Q-Q3) to avoid losing a piece by capitalizing on the immobility of the Black Queen Pawn.

19 . . .	Q-B3!
20 B-B4	Q × QBP
21 Q × Q	R × Q
22 R-QB1	

White is lost but relatively best was 22. R-Q3 blockading the passed Queen Pawn.
22. R-QB1 indicates that an order is missing to avoid exchanges after losing material, unless such exchanges deserve a high rating for specific reasons covered by other orders.

22	. . .	KR-QB1
23	QR-Q1	

White is just floundering in a lost position.

23	. . .	KR-B6
24	P-N4	

"There are no good moves in bad positions!"

24	. . .	KR × P
25	B-N3	

Best; White at least stops the mating attack.

25	. . .	P-Q6
26	R-QB1	B-N4

26. R-QB1 indicates that an order is missing that would make the machine avoid getting forked.
Better was 26 . . . , P-Q7 winning instantly (26 . . . , P-Q7; 27. R × R, P × R = Qch; 28. K-N2, Q-Q8!, 29. R-B8ch, B-Q1).

27	R × R	P × R
28	B-K5	P-B8 = Q
29	R × Q	B × R
30	Resigns	

Best, but I'm sure the programmers were just getting tired!
Such test games give indeed excellent indications as to the type of general principles the program should include in addition to material balance, development, and center control, to eliminate antipositional moves as much as possible.

SOME STUDIES IN MACHINE LEARNING USING THE GAME OF CHECKERS

by A. L. Samuel

Introduction

The studies reported here have been concerned with the programming of a digital computer to behave in a way which, if done by human beings or animals, would be described as involving the process of learning. While this is not the place to dwell on the importance of machine-learning procedures, or to discourse on the philosophical aspects,[1] there is obviously a very large amount of work, now done by people, which is quite trivial in its demands on the intellect but does, nevertheless, involve some learning. We have at our command computers with adequate data-handling ability and with sufficient computational speed to make use of machine-learning techniques, but our knowledge of the basic principles of these techniques is still rudimentary. Lacking such knowledge, it is necessary to specify methods of problem solution in minute and exact detail, a time-consuming and costly procedure. Programming computers to learn from experience should eventually eliminate the need for much of this detailed programming effort.

General Methods of Approach

At the outset it might be well to distinguish sharply between two general approaches to the problem of machine learning. One method, which might be called the *Neural-Net Approach,* deals with the possibility of inducing learned behavior into a randomly connected switching net (or its simula-

[1] Some of these are quite profound and have a bearing on the questions raised by Nelson Goodman in *Fact, Fiction and Forecast,* Cambridge, Mass.: Harvard, 1954.

tion on a digital computer) as a result of a reward-and-punishment routine. A second, and much more efficient approach, is to produce the equivalent of a highly organized network which has been designed to learn only certain specific things. The first method should lead to the development of general-purpose learning machines. A comparison between the size of the switching nets that can be reasonably constructed or simulated at the present time and the size of the neural nets used by animals, suggests that we have a long way to go before we obtain practical devices.[2] The second procedure requires reprogramming for each new application, but it is capable of realization at the present time. The experiments to be described here were based on this second approach.

Choice of Problem

For some years the writer has devoted his spare time to the subject of machine learning and has concentrated on the development of learning procedures as applied to games.[3] A game provides a convenient vehicle for such study as contrasted with a problem taken from life, since many of the complications of detail are removed. Checkers, rather than chess (Shannon, 1950; Bernstein and Roberts, 1958*b*; Kister et al., 1957; Newell, Shaw, and Simon, 1958*b*), was chosen because the simplicity of its rules permits greater emphasis to be placed on learning techniques. Regardless of the relative merits of the two games as intellectual pastimes, it is fair to state that checkers contains all of the basic characteristics of an intellectual activity in which heuristic procedures and learning processes can play a major role and in which these processes can be evaluated.

Some of these characteristics might well be enumerated. They are:

(1) The activity must not be deterministic in the practical sense. There exists no known algorithm which will guarantee a win or a draw in checkers, and the complete explorations of every possible path through a checker game would involve perhaps 10^{40} choices of moves which, at 3 choices per millimicrosecond, would still take 10^{21} centuries to consider.

(2) A definite goal must exist—the winning of the game—and at least one criterion or intermediate goal must exist which has a bearing on the achievement of the final goal and for which the sign should be known. In checkers the goal is to deprive the opponent of the possibility of moving,

[2] Warren S. McCulloch (1949) has compared the digital computer to the nervous system of a flatworm. To extend this comparison to the situation under discussion would be unfair to the worm, since its nervous system is actually quite highly organized as compared with the random-net studies by Farley and Clark (1954), Rochester, Holland, Haibt, and Duda (1956), and by Rosenblatt (1958).

[3] The first operating checker program for the IBM 701 was written in 1952. This was recoded for the IBM 704 in 1954. The first program with learning was completed in 1955 and demonstrated on television on February 24, 1956.

and the dominant criterion is the number of pieces of each color on the board. The importance of having a known criterion will be discussed later.

(3) The rules of the activity must be definite and they should be known. Games satisfy this requirement. Unfortunately, many problems of economic importance do not. While in principle the determination of the rules can be a part of the learning process, this is a complication which might well be left until later.

(4) There should be a background of knowledge concerning the activity against which the learning progress can be tested.

(5) The activity should be one that is familiar to a substantial body of people so that the behavior of the program can be made understandable to them. The ability to have the program play against human opponents (or antagonists) adds spice to the study and, incidentally, provides a convincing demonstration for those who do not believe that machines can learn.

Having settled on the game of checkers for our learning studies, we must, of course, first program the computer to play legal checkers; that is, we must express the rules of the game in machine language and we must arrange for the mechanics of accepting an opponent's moves and of reporting the computer's moves, together with all pertinent data desired by the experimenter. The general methods for doing this were described by Shannon in 1950 as applied to chess rather than checkers. The basic program used in these experiments is quite similar to the program described by Strachey in 1952. The availability of a larger and faster machine (the IBM 704), coupled with many detailed changes in the programming procedure, leads to a fairly interesting game, even without any learning. The basic forms of the program will now be described.

The Basic Checker-playing Program

The computer plays by looking ahead a few moves and by evaluating the resulting board positions much as a human player might do. Board positions are stored by sets of machine words, four words normally being used to represent any particular board position. Thirty-two bit positions (of the 36 available in an IBM 704 word) are, by convention, assigned to the 32 playing squares on the checkerboard, and pieces appearing on these squares are represented by 1's appearing in the assigned bit positions of the corresponding word. "Looking ahead" is prepared for by computing all possible next moves, starting with a given board position. The indicated moves are explored in turn by producing new board-position records corresponding to the conditions after the move in question (the old board positions being saved to facilitate a return to the starting point) and the process can be repeated. This look-ahead procedure is carried several

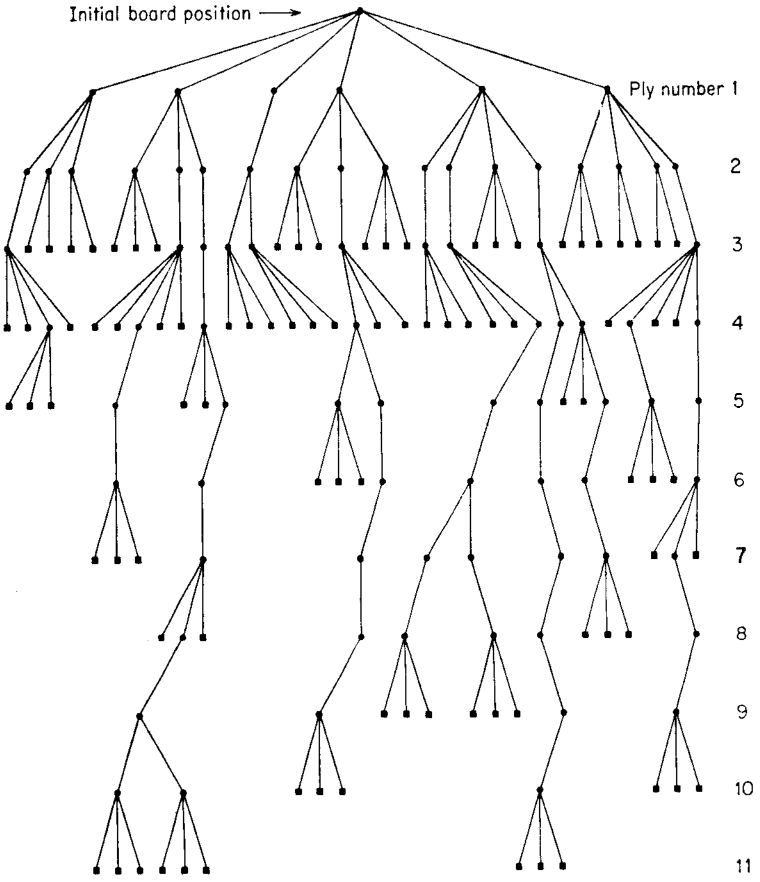


Figure 1. A "tree" of moves which might be investigated during the look-ahead procedure. The actual branchings are much more numerous than those shown, and the "tree" is apt to extend to as many as 20 levels.

moves in advance, as illustrated in Fig. 1. The resulting board positions are then scored in terms of their relative value to the machine.

The standard method of scoring the resulting board positions has been in terms of a linear polynomial. A number of schemes of an abstract sort were tried for evaluating board positions without regard to the usual checker concepts, but none of these was successful.[4] One way of looking at the various terms in the scoring polynomial is that those terms with

[4] One of the more interesting of these was to express a board position in terms of the first and higher moments of the white and black pieces separately about two orthogonal axes on the board. Two such sets of axes were tried, one set being parallel to the sides of the board and the second set being those through the diagonals

numerically small coefficients should measure criteria related as intermediate goals to the criteria measured by the larger terms. The achievement of these intermediate goals indicates that the machine is going in the right direction, such that the larger terms will eventually increase. If the program could look far enough ahead we need only ask, "Is the machine still in the game?"[5] Since it cannot look this far ahead in the usual situation, we must substitute something else, say the piece ratio, and let the machine continue the look-ahead until one side has gained a piece advantage. But even this is not always possible, so we have the program test to see if the machine has gained a positional advantage, et cetera. Numerical measures of these various properties of the board positions are then added together (each with an appropriate coefficient which defines its relative importance) to form the evaluation polynomial.

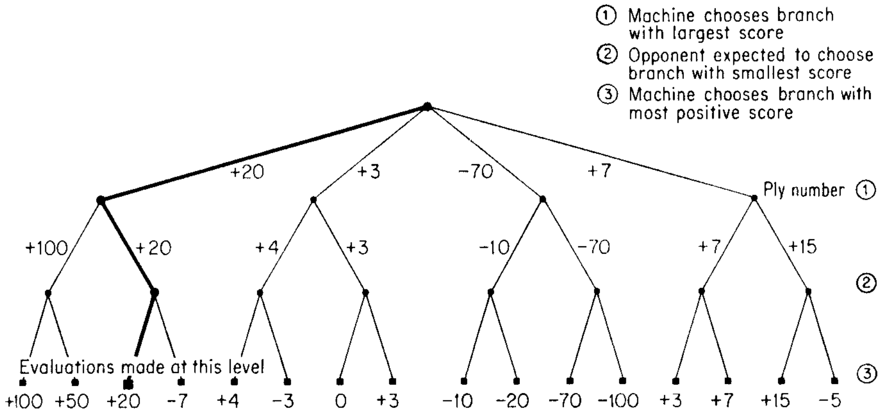
More specifically, as defined by the rules for checkers, the dominant scoring parameter is the inability for one side or the other to move.[6] Since this can occur but once in any game, it is tested for separately and is not included in the scoring polynomial as tabulated by the computer during play. The next parameter to be considered is the relative piece advantage. It is always assumed that it is to the machine's advantage to reduce the number of the opponent's pieces as compared to its own. A reversal of the sign of this term will, in fact, cause the program to play "giveaway" checkers, and with learning it can only learn to play a better and better giveaway game. Were the sign of this term not known by the programmer it could, of course, be determined by tests, but it must be fixed by the experimenter and, in effect, it is one of the instructions to the machine defining its task. The numerical computation of the piece advantage has been arranged in such a way as to account for the well-known property that it is usually to one's advantage to trade pieces when one is ahead and to avoid trades when behind. Furthermore, it is assumed that kings are more valuable than pieces, the relative weights assigned to them being three to two.[7] This ratio means that the program will trade three men for two kings, or two kings for three men, if by so doing it can obtain some positional advantage.

The choice for the parameters to follow this first term of the scoring polynomial and their coefficients then becomes a matter of concern. Two courses are open—either the experimenter can decide what these subsequent terms are to be, or he can arrange for the program to make the selection. We will discuss the first case in some detail in connection with

[5] This apt phraseology was suggested by John McCarthy.

[6] Not the capture of all the opponent's pieces, as popularly assumed, although all games end in this fashion.

[7] The use of a weight ratio rather than this, conforming more closely to the values assumed by many players, can lead into certain logical complications, as found by Strachey (1952).



① Machine chooses branch with largest score
② Opponent expected to choose branch with smallest score
③ Machine chooses branch with most positive score
Ply number ①
②
③
+20
+3
-70
+7
+100
+20
+4
+3
-10
-70
+7
+15
Evaluations made at this level
+100 +50 +20 -7 +4 -3 0 +3 -10 -20 -70 -100 +3 +7 +15 -5

Figure 2. Simplified diagram showing how the evaluations are backed up through the "tree" of possible moves to arrive at the best next move. The evaluation process starts at (3).

the rote-learning studies and leave for a later section the discussion of various program methods of selecting parameters and adjusting their coefficients.

It is not satisfactory to select the initial move which leads to the board position with the highest score, since to reach this position would require the cooperation of the opponent. Instead, an analysis must be made proceeding *backward* from the evaluated board positions through the "tree" of possible moves, each time with consideration of the intent of the side whose move is being examined, assuming that the opponent would always attempt to minimize the machine's score while the machine acts to maximize its score. At each branch point, then, the corresponding board position is given the score of the board position which would result from the most favorable move. Carrying this "minimax" procedure back to the starting point results in the selection of a "best move." The score of the board position at the end of the most likely chain is also brought back, and for learning purposes this score is now assigned to the present board position. This process is shown in Fig. 2. The best move is executed, reported on the console lights, and tabulated by the printer.

The opponent is then permitted to make his move, which can be communicated to the machine either by means of console switches or by means of punched cards. The computer verifies the legality of the opponent's move, rejecting[8] or accepting it, and the process is repeated. When the program can look ahead and predict a win, this fact is reported on the

[8] The only departure from complete generality of the game as programmed is that the program requires the opponent to make a permissible move, including the taking of a capture if one is offered. "Huffing" is not permitted.

printer. Similarly, the program concedes when it sees that it is going to lose.

Ply Limitations

Playing-time considerations make it necessary to limit the look-ahead distance to some fairly small value. This distance is defined as the *ply* (a ply of 2 consisting of one proposed move by the machine and the anticipated reply by the opponent). The ply is not fixed but depends upon the dynamics of the situation, and it varies from move to move and from branch to branch during the move analysis. A great many schemes of adjusting the look-ahead distance have been tried at various times, some of them quite complicated. The most effective one, although quite detailed, is simple in concept and is as follows. The program always looks ahead a minimum distance, which for the opening game and without learning is usually set at three moves. At this minimum ply the program will evaluate the board position if none of the following conditions occurs: (1) the next move is a jump, (2) the last move was a jump, or (3) an exchange offer is possible. If any one of these conditions exists, the program continues looking ahead. At a ply of 4 the program will stop and evaluate the resulting board position if conditions (1) and (3) above are not met. At a ply of 5 or greater, the program stops the look-ahead whenever the next ply level does not offer a jump. At a ply of 11 or greater, the program will terminate the look-ahead, even if the next move is to be a jump, should one side at this time be ahead by more than two kings (to prevent the needless exploration of obviously losing or winning sequences). The program stops at a ply of 20 regardless of all conditions (since the memory space for the look-ahead moves is then exhausted) and an adjustment in score is made to allow for the pending jump. Finally, an adjustment is made in the levels of the break points between the different conditions when time is saved through rote learning (see below) and when the total number of pieces on the board falls below an arbitrary number. All break points are determined by single data words which can be changed at any time by manual intervention.

This tying of the ply with board conditions achieves three desired results. In the first place, it permits board evaluations to be made under conditions of relative stability for so-called dead positions, as defined by Turing (Bowden, 1953). Secondly, it causes greater surveillance of those paths which offer better opportunities for gaining or losing an advantage. Finally, since branching is usually seriously restricted by a jump situation, the total number of board positions and moves to be considered is still held down to a reasonable number and is more equitably distributed between the various possible initial moves.

As a practical matter, machine playing time usually has been limited to approximately 30 seconds per move. Elaborate table look-up procedures, fast sorting and searching procedures, and a variety of new programming tricks were developed, and full use was made of all of the resources of the IBM 704 to increase the operating speed as much as possible. One can, of course, set the playing time at any desired value by adjustments of the permitted ply; too small a ply results in a bad game and too large a ply makes the game unduly costly in terms of machine time.

Other Modes of Play

For study purposes the program was written to accommodate several variations of this basic plan. One of these permits the program to play against itself, that is, to play both sides of the game. This mode of play has been found to be especially good during the early stages of learning.

The program can also follow book games presented to it either on cards or on magnetic tape. When operating in this mode, the program decides at each point in the game on its next move in the usual way and reports this proposed move. Instead of actually making this move, the program refers to the stored record of a book game and makes the book move. The program records its evaluation of the two moves, and it also counts and reports the number of possible moves which the program rates as being better than the book move and the number it rates as being poorer. The sides are then reversed and the process is repeated. At the end of a book game a correlation coefficient is computed, relating the machine's indicated moves to those moves adjudged best by the checker masters.[9]

It should be noted that the emphasis throughout all of these studies has been on learning techniques. The temptation to improve the machine's game by giving it standard openings or other man-generated knowledge of playing techniques has been consistently resisted. Even when book games are played, no weight is given to the fact that the moves as listed are presumably the best possible moves under the circumstances.

For demonstration purposes, and also as a means of avoiding lost machine time while an opponent is thinking, it is sometimes convenient to play several simultaneous games against different opponents. With the program in its present form the most convenient number for this purpose has been found to be six, although eight have been played on a number of occasions.

Games may be started with any initial configuration for the board position so that the program may be tested on end games, checker puzzles, et cetera. For nonstandard starting conditions, the program lists the initial

[9] This coefficient is defined as $C = (L - H)/(L + H)$, where L is the total number of different legal moves which the machine judged to be poorer than the indicated book moves, and H is the total number which it judged to be better than the book moves.

piece arrangement. From time to time, and at the end of each game, the program also tabulates various bits of statistical information which assist in the evaluation of playing performance.

Numerous other features have also been added to make the program convenient to operate (for details see Appendix A), but these have no direct bearing on the problem of learning, to which we will now turn our attention.

Rote Learning and Its Variants

Perhaps the most elementary type of learning worth discussing would be a form of rote learning in which the program simply saved all of the board positions encountered during play, together with their computed scores. Reference could then be made to this memory record and a certain amount of computing time might be saved. This can hardly be called a very advanced form of learning; nevertheless, if the program then utilizes the saved time to compute further in depth it will improve with time.

Fortunately, the ability to store board information at a ply of 0 and to look up boards at a larger ply provides the possibility of looking much farther in advance than might otherwise be possible. To understand this, consider a very simple case where the look ahead is always terminated at a fixed ply, say 3. Assume further that the program saves only the board positions encountered during the actual play with their associated backed-up scores. Now it is this list of previous board positions that is used to look up board positions while at a ply level of 3 in the subsequent games. If a board position is found, its score has, in effect, already been backed up by three levels, and if it becomes effective in determining the move to be made, it is a 6-ply score rather than a simple 3-ply score. This new initial board position with its 6-ply score is, in turn, saved and it may be encountered in a future game and the score backed up by an additional set of three levels, et cetera. This procedure is illustrated in Fig. 3. The incorporation of this variation, together with the simpler rote-learning feature, results in a fairly powerful learning technique which has been studied in some detail.

Several additional features had to be incorporated into the program before it was practical to embark on learning studies using this storage scheme. In the first place, it was necessary to impart a sense of direction to the program in order to force it to press on toward a win. To illustrate this, consider the situation of two kings against one king, which is a winning combination for practically all variations in board positions. In time, the program can be assumed to have stored all of these variations, each associated with a winning score. Now, if such a situation is encountered, the program will look ahead along all possible paths and each path will

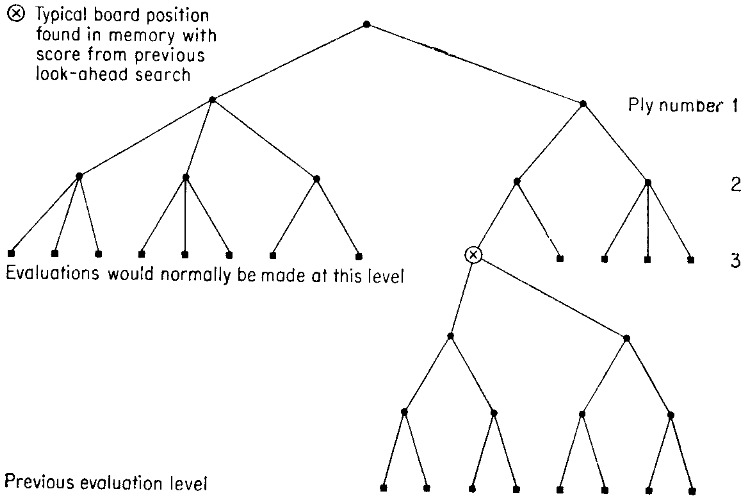

⊗ Typical board position found in memory with score from previous look-ahead search
Ply number 1
2
3
Evaluations would normally be made at this level
Previous evaluation level

Figure 3. Simplified representation of the rote-learning process, in which information saved from a previous game is used to increase the effective ply of the backed-up score.

lead to a winning combination, in spite of the fact that only one of the possible initial moves may be along the direct path toward the win while all of the rest may be wasting time. How is the program to differentiate between these?

A good solution is to keep a record of the ply value of the different board positions at all times and to make a further choice between board positions on this basis. If ahead, the program can be arranged to push directly toward the win while, if behind, it can be arranged to adopt delaying tactics. The most recent method used is to carry the effective ply along with the score by simply decreasing the magnitude of the score a small amount each time it is backed up a ply level during the analyses. If the program is now faced with a choice of board positions whose scores differ only by the ply number, it will automatically make the most advantageous choice, choosing a low-ply alternative if winning and a high-ply alternative if losing. The significance of this concept of a direction sense should not be overlooked. Even without "learning," it is very important. Several of the early attempts at learning failed because the direction sense was not properly taken into account.

Cataloging and Culling Stored Information

Since practical considerations limit the number of board positions which can be saved, and since the time to search through those that are saved can easily become unduly long, one must devise systems (1) to catalog boards that are saved, (2) to delete redundancies, and (3) to discard board positions which are not believed to be of much value. The most effective cataloging system found to date starts by standardizing all board positions, first by reversing the pieces and piece positions if it is a board position in which White is to move, so that all boards are reported as if it were Black's turn to move. This reduces by nearly a factor of two the number of boards which must be saved. Board positions, in which all of the pieces are kings, can be reflected about the diagonals with a possible fourfold reduction in the number which must be saved. A more compact board representation than the one employed during play is also used so as to minimize the storage requirements.

After the board positions are standardized, they are grouped into records on the basis of (1) the number of pieces on the board, (2) the presence or absence of a piece advantage, (3) the side possessing this advantage, (4) the presence or absence of kings on the board, (5) the side having the so-called "move," or opposition advantage, and finally (6) the first moments of the pieces about normal and diagonal axes through the board. During play, newly acquired board positions are saved in the memory until a reasonable number have been accumulated, and they are then merged with those on the "memory tape" and a new memory tape is produced. Board positions within a record are listed in a serial fashion, being sorted with respect to the words which define them. The records are arranged on the tape in the order that they are most likely to be needed during the course of a game; board positions with 12 pieces to a side coming first, et cetera. This method of cataloging is very important because it cuts tape-searching time to a minimum.

Reference must be made, of course, to the board positions already saved, and this is done by reading the correct record into the memory and searching through it by a dichotomous search procedure. Usually five or more records are held in memory at one time, the exact number at any time depending upon the lengths of the particular records in question. Normally, the program calls three or four new records into memory during each new move, making room for them as needed, by discarding the records which have been held the longest.

Two different procedures have been found to be of value in limiting the number of board positions that are saved; one based on the frequency of use, and the second on the ply. To keep track of the frequency of use, an age term is carried along with the score. Each new board position to be saved is arbitrarily assigned an age. When reference is made to a stored board position, either to update its score or to utilize it in the look-ahead procedure, the age recorded for this board position is divided by two. This is called *refreshing.* Offsetting this, each board position is automatically aged by one unit at the memory merge times (normally occurring about once every 20 moves). When the age of any one board position reaches an arbitrary maximum value this board position is expunged from the record. This is a form of *forgetting.* New board positions which remain unused are soon forgotten, while board positions which are used several times in succession will be refreshed to such an extent that they will be remembered even if not used thereafter for a fairly long period of time. This form of refreshing and forgetting was adopted on the basis of reflections as to the frailty of human memories. It has proven to be very effective.

In addition to the limitations imposed by forgetting, it seemed desirable to place a restriction on the maximum size of any one record. Whenever an arbitrary limit is reached, enough of the lowest-ply board positions are automatically culled from the record to bring the size well below the maximum.

Before embarking on a study of the learning capabilities of the system as just described, it was, of course, first necessary to fix the terms and coefficients in the evaluation polynomial. To do this, a number of different sets of values were tested by playing through a series of book games and computing the move correlation coefficients. These values varied from 0.2 for the poorest polynomial tested, to approximately 0.6 for the one finally adopted. The selected polynomial contained four terms (as contrasted with the use of 16 terms in later experiments). In decreasing order of importance these were: (1) piece advantage, (2) denial of occupancy, (3) mobility, and (4) a hybrid term which combined control of the center and piece advancement.

Rote-learning Tests

After a scoring polynomial was arbitrarily picked, a series of games was played, both self-play and play against many different individuals (several of these being checker masters). Many book games were also followed, some of these being end games. The program learned to play a very good opening game and to recognize most winning and losing end positions many moves in advance, although its midgame play was not greatly improved. This program now qualifies as a rather better-than-average novice, but definitely not as an expert.

At the present time the memory tape contains something over 53,000 board positions (averaging 3.8 words each) which have been selected from a much larger number of positions by means of the culling techniques described. While this is still far from the number which would tax the listing and searching procedures used in the program, rough estimates, based on the frequency with which the saved boards are utilized during normal play (these figures being tabulated automatically), indicate that a library tape containing at least 20 times the present number of board positions would be needed to improve the midgame play significantly. At the present rate of acquisition of new positions this would require an inordinate amount of play and, consequently, of machine time.[10]

The general conclusions which can be drawn from these tests are that:

(1) An effective rote-learning technique must include a procedure to give the program a sense of direction, and it must contain a refined system for cataloging and storing information.

(2) Rote-learning procedures can be used effectively on machines with the data-handling capacity of the IBM 704 if the information which must be saved and searched does not occupy more than, roughly, one million words, and if not more than one hundred or so references need to be made to this information per minute. These figures are, of course, highly dependent upon the exact efficiency of cataloging which can be achieved.

(3) The game of checkers, when played with a simple scoring scheme and with rote learning only, requires more than this number of words for master caliber of play and, as a consequence, is not completely amenable to this treatment on the IBM 704.

(4) A game, such as checkers, is a suitable vehicle for use during the development of learning techniques, and it is a very satisfactory device for demonstrating machine learning procedures to the unbelieving.

Learning Procedure Involving Generalizations

An obvious way to decrease the amount of storage needed to utilize past experience is to generalize on the basis of experience and to save only the generalizations. This should, of course, be a continuous process if it is to be truly effective, and it should involve several levels of abstraction. A start has been made in this direction by having the program select a subset of possible terms for use in the evaluation polynomial and by having the program determine the sign and magnitude of the coefficients which multiply these parameters. At the present time this subset consists of 16 terms chosen from a list of 38 parameters. The piece-advantage term needed to define the task is computed separately and, of course, is not altered by the program.

After a number of relatively unsuccessful attempts to have the program generalize while playing both sides of the game, the program was arranged to act as two different players, for convenience called *Alpha* and *Beta*. Alpha generalizes on its experience after each move by adjusting the coefficients in its evaluation polynomial and by replacing terms which appear to be unimportant by new parameters drawn from a reserve list. Beta, on the contrary, uses the same evaluation polynomial for the dura-

[10] This playing-time requirement, while large in terms of cost, would be less than the time which the checker master probably spends to acquire his proficiency.

tion of any one game. Program Alpha is used to play against human opponents, and during self-play Alpha and Beta play each other.

At the end of each self-play game a determination is made of the relative playing ability of Alpha, as compared with Beta, by a neutral portion of the program. If Alpha wins—or is adjudged to be ahead when a game is otherwise terminated—the then current scoring system used by Alpha is given to Beta. If, on the other hand, Beta wins or is ahead, this fact is recorded as a black mark for Alpha. Whenever Alpha receives an arbitrary number of black marks (usually set at three) it is assumed to be on the wrong track, and a fairly drastic and arbitrary change is made in its scoring polynomial (by reducing the coefficient of the leading term to zero). This action is necessary on occasion, since the entire learning process is an attempt to find the highest point in multidimensional scoring space in the presence of many secondary maxima on which the program can become trapped. By manual intervention it is possible to return to some previous condition or make some other change if it becomes apparent that the learning process is not functioning properly. In general, however, the program seeks to extricate itself from traps and to improve more or less continuously.

The capability of the program can be tested at any time by having Alpha play one or more book games (with the learning procedure temporarily immobilized) and by correlating its play with the recommendations of the masters or, more interestingly, by pitting it against a human player.

Polynomial Modification Procedure

If Alpha is to make changes in its scoring polynomial, it must be given some trustworthy criteria for measuring performance. A logical difficulty presents itself, since the only measuring parameter available is this same scoring polynomial that the process is designed to improve. Recourse is had to the peculiar property of the look-ahead procedure, which makes it less important for the scoring polynomial to be particularly good the further ahead the process is continued. This means that one can evaluate the relative change in the positions of two players, when this evaluation is made over a fairly large number of moves, by using a scoring system which is much too gross to be significant on a move-by-move basis.

Perhaps an even better way of looking at the matter is that we are attempting to make the score, calculated for the current board position, look like that calculated for the terminal board position of the chain of moves which most probably will occur during actual play. Of course, if one could develop a perfect system of this sort it would be the equivalent of always looking ahead to the end of the game. The nearer this ideal is approached, the better would be the play.[11]

[11] There is a logical fallacy in this argument. The program might save only invariant terms which have nothing to do with goodness of play; for example, it might

In order to obtain a sufficiently large span to make use of this characteristic, Alpha keeps a record of the apparent goodness of its board positions as the game progresses. This record is kept by computing the scoring polynomial for each board position encountered in actual play and by saving this polynomial in its entirety. At the same time, Alpha also computes the backed-up score for all board positions, using the look-ahead procedure described earlier. At each play by Alpha the initial board score, as saved from the previous Alpha move, is compared with the backed-up score for the current position. The difference between these scores, defined as *delta,* is used to check the scoring polynomial. If delta is positive it is reasonable to assume that the initial board evaluation was in error and terms which contributed positively should have been given more weight, while those that contributed negatively should have been given less weight. A converse statement can be made for the case where delta is negative. Presumably, in this case, either the initial board evaluation was incorrect, or a wrong choice of moves was made, and greater weight should have been given to terms making negative contributions, with less weight to positive terms. These changes are not made directly but are brought about in an involved way which will now be described.

A record is kept of the correlation existing between the signs of the individual term contributions in the initial scoring polynomial and the sign of delta. After each play an adjustment is made in the values of the correlation coefficients, due account being taken of the number of times that each particular term has been used and has had a nonzero value. The coefficient for the polynomial term (other than the piece-advantage term) with the then largest correlation coefficient is set at a prescribed maximum value with proportionate values determined for all of the remaining coefficients. Actually, the term coefficients are fixed at integral powers of 2, this power being defined by the ratio of the correlation coefficients. More precisely, if the ratio of two correlation coefficients is equal to or larger than n but less than $n + 1$, where n is an integer, then the ratio of the two term coefficients is set equal to 2^n. This procedure was adopted in order to increase the range in values of the term coefficients. Whenever a correlation-coefficient calculation leads to a negative sign, a corresponding reversal is made in the sign associated with the term itself.

Instabilities

It should be noted that the span of moves over which delta is computed consists of a remembered part and an anticipated portion. During the remembered play, use had been made of Alpha's current scoring polynomial to determine Alpha's moves but not to determine the opponent's moves,

count the squares on the checkerboard. The forced inclusion of the piece-advantage term prevents this.

while during the anticipation play the moves for both sides are made using Alpha's scoring polynomial. One is tempted to increase the sensitivity of delta as an indicator of change by increasing the span of the remembered portion. This has been found to be dangerous since the coefficients in the evaluation polynomial and, indeed, the terms themselves, may change between the time of the remembered evaluation and the time at which the anticipation evaluation is made. As a matter of fact, this difficulty is present even for a span of one move pair. It is necessary to recompute the scoring polynomial for a given initial board position after a move has been determined and after the indicated corrections in the scoring polynomial have been made, and to save this score for future comparisons, rather than to save the score used to determine the move. This may seem a trivial point, but its neglect in the initial stages of these experiments led to oscillations quite analogous to the instability induced in electrical circuits by long delays in a feedback loop.

As a means of stabilizing against minor variations in the delta values, an arbitrary minimum value was set, and when delta fell below this minimum for any particular move no change was made in the polynomial. This same minimum value is used to set limits for the initial board evaluation score to decide whether or not it will be assumed to be zero. This minimum is recomputed each time and, normally, has been fixed at the average value of the coefficients for the terms in the currently existing evaluation polynomial.

Still another type of instability can occur whenever a new term is introduced into the scoring polynomial. Obviously, after only a single move the correlation coefficient of this new term will have a magnitude of 1, even though it might go to 0 after the very next move. To prevent violent fluctuations due to this cause, the correlation coefficients for newly introduced terms are computed as if these terms had already been used several times and had been found to have a zero correlation coefficient. This is done by replacing the times-used number in the calculation by an arbitrary number (usually set at 16) until the usage does, in fact, equal this number.

After a term has been in use for some time, quite the opposite action is desired so that the more recent experience can outweigh earlier results. This is achieved, together with a substantial reduction in calculation time, by using powers of 2 in place of the actual times used and by limiting the maximum power that is used. To be specific, at any stage of play defined as the Nth move, corrections to the values of the correlation coefficients C_N are made using 16 for N until N equals 32, whereupon 32 is used until N equals 64, et cetera, using the formula:

$$C_N = C_{N-1} - \frac{C_{N-1} \pm 1}{N}$$

and a value for N larger than 256 is never used.

After a minimum was set for delta it seemed reasonable to attach greater weight to situations leading to large values of delta. Accordingly, two additional categories are defined. If a contribution to delta is made by the first term, meaning that a change has occurred in the piece ratio, the indicated changes in the correlation coefficients are doubled, while if the value of delta is so large as to indicate that an almost sure win or lose will result, the effect on the correlation coefficients is quadrupled.

Term Replacement

Mention has been made several times of the procedure for replacing terms in the scoring polynomial. The program, as it is currently running, contains 38 different terms (in addition to the piece-advantage term), 16 of these being included in the scoring polynomial at any one time and the remaining 22 being kept in reserve. After each move a low-term tally is recorded against that active term which has the lowest correlation coefficient and, at the same time, a test is made to see if this brings its tally count up to some arbitrary limit, usually set at 8. When this limit is reached for any specific term, this term is transferred to the bottom of the reserve list, and it is replaced by a term from the head of the reserve list. This new term enters the polynomial with zero values for its correlation coefficient, times used, and low-tally count. On the average, then, an active term is replaced once each eight moves and the replaced terms are given another chance after 176 moves. As a check on the effectiveness of this procedure, the program reports on the usage which has accrued against each discarded term. Terms which are repeatedly rejected after a minimum amount of usage can be removed and replaced with completely new terms.

It might be argued that this procedure of having the program select terms for the evaluation polynomial from a supplied list is much too simple and that the program should generate the terms for itself. Unfortunately, no satisfactory scheme for doing this has yet been devised. With a man-generated list one might at least ask that the terms be members of an orthogonal set, assuming that this has some meaning as applied to the evaluation of a checker position. Apparently, no one knows enough about checkers to define such a set. The only practical solution seems to be that of including a relatively large number of possible terms in the hope that all of the contributing parameters get covered somehow, even though in an involved and redundant way. This is not an undesirable state of affairs, however, since it simulates the situation which is likely to exist when an attempt is made to apply similar learning techniques to real-life situations.

Many of the terms in the existing list are related in some vague way to the parameters used by checker experts. Some of the concepts which checker experts appear to use have eluded the writer's attempts at definition, and he has been unable to program them. Some of the terms are quite unrelated to the usual checker lore and have been discovered more or less by accident. The second moment about the diagonal axis through the double corners is an example. Twenty-seven different simple terms are now in use, the rest being combinational terms, as will be described later.

A word might be said about these terms with respect to the exact way in which they are defined and the general procedures used for their evaluation. Each term relates to the relative standings of the two sides, with respect to the parameter in question, and it is numerically equal to the difference between the ratings for the individual sides. A reversal of the sign obviously corresponds to a change of sides. As a further means of insuring symmetry the individual ratings of the respective sides are determined at corresponding times in the play as viewed by the side in question. For example, consider a parameter which relates to the board conditions as left after one side has moved. The rating of Black for such a parameter would be made after Black had moved, and the rating of White would not be made until after White had moved. During anticipation play, these individual ratings are made after each move and saved for future reference. When an evaluation is desired the program takes the differences between the most recent ratings and those made a move earlier. In general, an attempt has been made to define all parameters so that the individual-side ratings are expressible as small positive integers.

Binary Connective Terms

In addition to the simple terms of the type just described, a number of combinational terms have been introduced. Without these terms the scoring polynomial would, of course, be linear. A number of different ways of introducing nonlinear terms have been devised but only one of these has been tested in any detail. This scheme provides terms which have some of the properties of binary logical connectives. Four such terms are formed for each pair of simple terms which are to be related. This is done by making an arbitrary division of the range in values for each of the simple terms and assigning the binary values of 0 and 1 to these ranges. Since most of the simple terms are symmetrical about 0, this is easily done on a sign basis. The new terms are then of the form $A \cdot B$, $A \cdot \bar{B}$, $\bar{A} \cdot B$, and $\bar{A} \cdot \bar{B}$, yielding values either of 0 or 1. These terms are introduced into the scoring polynomial with adjustable coefficients and signs, and are thereafter indistinguishable from the other terms.

As it would require some 1404 such combinational terms to interrelate the 27 simple terms originally used, it was found desirable to limit the actual number of combinational terms used at any one time to a small fraction of these and to introduce new terms only as it became possible to retire older ineffectual terms. The terms actually used are given in Appendix C.

Preliminary Learning-by-generalization Tests

An idea of the learning ability of this procedure can be gained by analyzing an initial test series of 28 games[12] played with the program just described. At the start an arbitrary selection of 16 terms was chosen and all terms were assigned equal weights. During the first 14 games Alpha was assigned the White side, with Beta constrained as to its first move (two cycles of the seven different initial moves). Thereafter, Alpha was assigned Black and White alternately. During this time a total of 29 different terms was discarded and replaced, the majority of these on two different occasions.

Certain other figures obtained during these 28 games are of interest. At frequent intervals the program lists the 12 leading terms in Alpha's scoring polynomial with their correlation coefficients and a running count of the number of times these coefficients have been altered. Based on these samplings, one observes that at least 20 different terms were assigned the largest coefficient at some time or other, some of these alternating with other terms a number of times, and two even reappearing at the top of the list with their signs reversed. While these variations were more violent at the start of the series of games and decreased as time went on, their presence indicated that the learning procedure was still not completely stable. During the first seven games there were at least 14 changes in occupancy at the top of the list involving 10 different terms. Alpha won three of these games and lost four. The quality of the play was extremely poor. During the next seven games there were at least eight changes made in the top listing involving five different terms. Alpha lost the first of these games and won the next six. Quality of play improved steadily but the machine still played rather badly. During Games 15 through 21 there were eight changes in the top listing involving five terms; Alpha winning five games and losing two. Some fairly good amateur players who played the machine during this period agreed that it was "tricky but beatable." During Games 22 through 28 there were at least four changes involving three terms. Alpha won two games and lost five. The program appeared to be approaching a quality of play which caused it to be described as "a better-than-average player." A detailed analysis of these results indicated that the learning procedure did work and that the rate of learning was surprisingly high, but that the learning was quite erratic and none too stable.

Second Series of Tests

Some of the more obvious reasons for this erratic behavior in the first series of tests have been identified. The program was modified in several

[12] The games averaged 68 moves (34 to a side) of which approximately 20 caused changes to be made in the scoring polynomial.

respects to improve the situation, and additional tests were made. Four of these modifications are important enough to justify a detailed explanation.

In the first place, the program was frequently fooled by bad play on the part of its opponent. A simple solution was to change the correlation coefficients less drastically when delta was positive than when delta was negative. The procedure finally adopted for the positive delta case was to make corrections to selected terms in the polynomial only. When the scoring polynomial was positive, changes were made to coefficients associated with the negatively contributing terms, and when the polynomial was negative, changes were made to the coefficients associated with positively contributing terms. No changes were made to coefficients associated with terms which happened to be zero. For the negative delta case, changes were made to the coefficients of all contributing terms, just as before.

A second defect seemed to be connected with the too frequent introduction of new terms into the scoring polynomial and the tendency for these new terms to assume dominant positions on the basis of insufficient evidence. This was remedied by the simple expedient of decreasing the rate of introduction of new terms from one every eight moves to one every 32 moves.

The third defect had to do with the complete exclusion from consideration of many of the board positions encountered during play by reason of the minimum limit on delta. This resulted in the misassignment of credit to those board positions which permitted spectacular moves when the credit rightfully belonged to earlier board positions which had permitted the necessary ground-laying moves. Although no precise way has yet been devised to ensure the correct assignment of credit, a very simple expedient was found to be most effective in minimizing the adverse effects of earlier assignments. This expedient was to allow the span of remembered moves, over which delta is computed, to increase until delta exceeded the arbitrary minimum value, and then to apply the corrections to the coefficients as dictated by the terms in the retained polynomial for this earlier board position. In this case, the difficulty which was mentioned in the section on Instabilities in connection with an arbitrary increase in span, does not occur after each correction, since no changes are made in the coefficients of the scoring polynomial as long as delta is below the minimum value. Of course, whenever delta does exceed the minimum value the program must then recompute the initial scoring polynomial for the then current board position and so restart the procedure with a span of a single remembered move pair. This over-all procedure rectifies the defect of assigning credit to a board position that lies too far along the move chain, but it introduces the possibility of assigning credit to a board position that is not far enough along.

As a partial expedient to compensate for this newly introduced danger, a change was made in the initial board evaluation. Instead of evaluating the initial board positions directly, as was done before, a standard but rudimentary tree search (terminated after the first nonjump move) was used. Errors due to impending jump situations were eliminated by this procedure, and because of the greater accuracy of the evaluation it was possible to reduce the minimum delta limit by a small amount.

Finally, to avoid the danger of having Beta adopt Alpha's polynomial as a result of a chance win on Alpha's part (or perhaps a situation in which Alpha had allowed its polynomial to degenerate after an early or midgame advantage had been gained), it was decided to require a majority of wins on Alpha's part before Beta would adopt Alpha's scoring polynomial.

With these modifications, a new series of tests was made. In order to reduce the learning time, the initial selection of terms was made on the basis of the results obtained during the earlier tests, but no attention was paid to their previously assigned weights. In contrast with the earlier erratic behavior, the revised program appeared to be extremely stable, perhaps at the expense of a somewhat lower initial learning rate. The way in which the character of the evaluation polynomial altered as learning progressed is shown in Fig. 4.

The most obvious change in behavior was in regard to the relative number of games won by Alpha and the prevalence of draws. During the first 28 games of the earlier series Alpha won 16 and lost 12. The corresponding figures for the first 28 games of the new series were 18 won by Alpha, and four lost, with six draws. In all cases the games were terminated, if not finished, in 70 moves and a judgment made in terms of the final positions. Unfortunately, these figures are not strictly comparable because of the decreased frequency with which Beta adopted Alpha's polynomial during the second series, both by design and because a programming error immobilized the adoption procedure during part of the tests. Nevertheless, the great decrease in the number of losses and the prevalence of draws seemed to indicate that the learning process was much more stable. Some typical games from this second series are given in Appendix B.

As learning proceeds, it should become harder and harder for Alpha to improve its game, and one would expect the number of wins by Alpha to decrease with time. If secondary maxima in scoring space are encountered, one might even find situations in which Alpha wins less than half of the games. With Beta at such a maximum any minor change in Alpha's polynomial would result in a degradation of its play, and several oscillations about the maximum might occur before Alpha landed at a point which would enable it to beat Beta. Some evidence of this trend is discernible in the play, although many more games will have to be played before it can be observed with certainty.

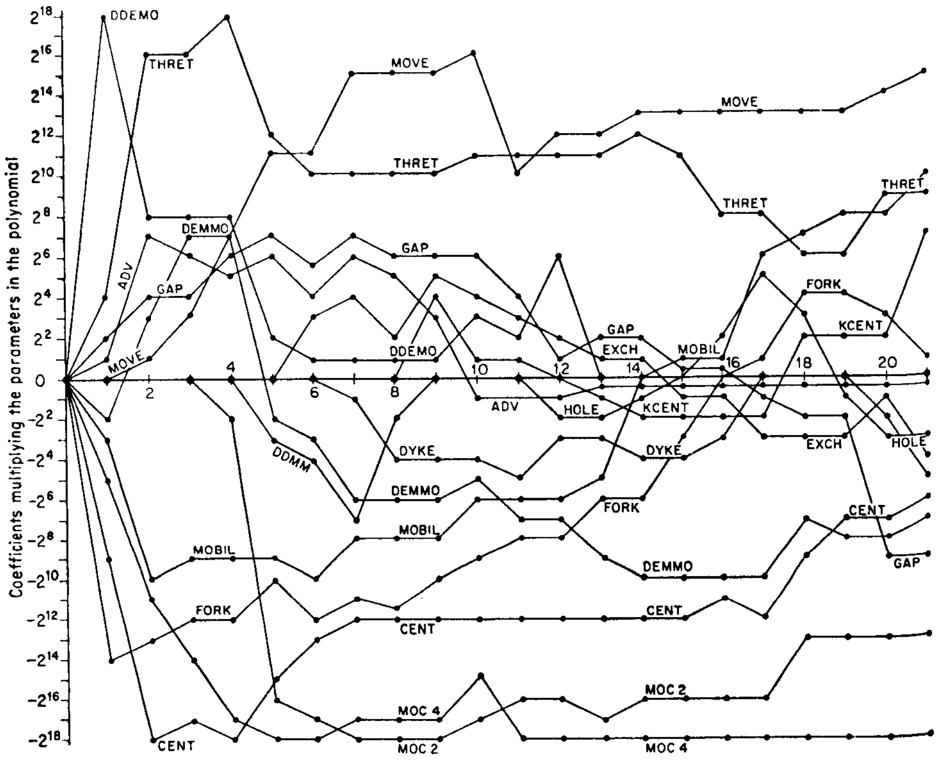


Figure 4. Second series of learning-by-generalization tests. Coefficients assigned by plotted as a function of the number of games played. Two regions of special found that the initial signs of many of the terms had been set incorrectly, and or 32 games.

The tentative conclusions which can be drawn from these tests are:

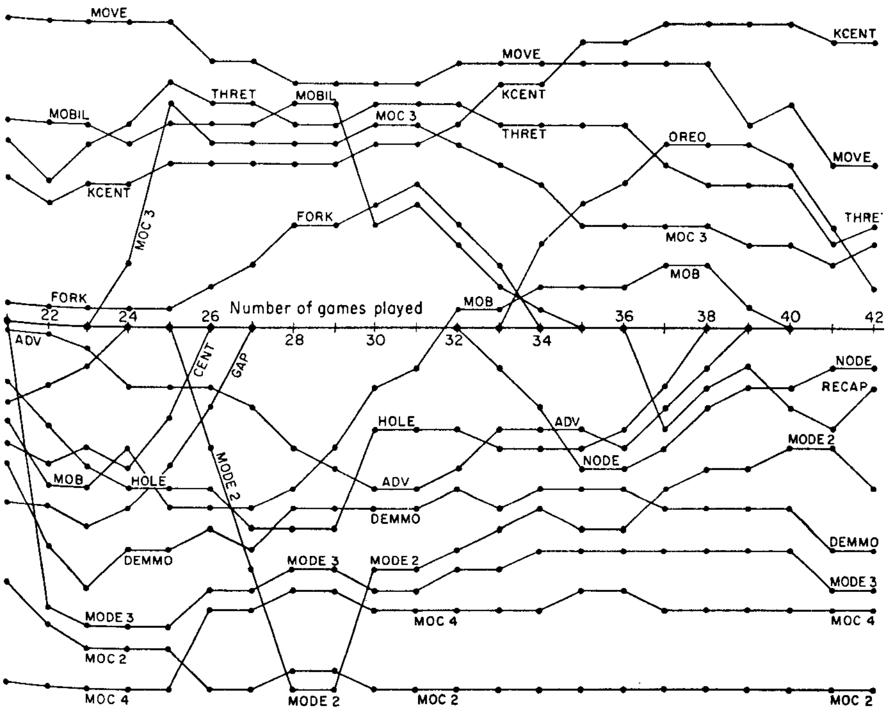
(1) A simple generalization scheme of the type here used can be an effective learning device for problems amenable to tree-searching procedures.

(2) The memory requirements of such schemes are quite modest and remain fixed with time.

(3) The operating times are also reasonable and remain fixed, independent of the amount of accumulated learning.

(4) Incipient forms of instability in the solution can be expected but, at least for the checker program, these can be dealt with by quite straightforward procedures.

(5) Even with the incomplete and redundant set of parameters which



the program to the more significant parameters of the evaluation polynomial
interest might be noted: (1) the situation after 13 or 14 games, when the program
(2) the conditions of relative stability which are beginning to show up after 31

have been used to date, it is possible for the computer to learn to play a better-than-average game of checkers in a relatively short period of time.

As a final precautionary note, it should be stated that these experiments have not encompassed a sufficiently large series of games to demonstrate unambiguously that the learning procedure is completely stable or that it will necessarily lead to the best possible choice of parameters and coefficients.

Rote Learning vs. Generalization

Some interesting comparisons can be made between the playing style developed by the learning-by-generalization program and that developed by the earlier rote-learning procedure. The program with rote learning soon learned to imitate master play during the opening moves. It was always quite poor during the middle game, but it easily learned how to avoid most of the obvious traps during end-game play and could usually drive on toward a win when left with a piece advantage. The program with the generalization procedure has never learned to play in a conventional manner and its openings are apt to be weak. On the other hand, it soon learned to play a good middle game, and with a piece advantage it usually polishes off its opponent in short order. Interestingly enough, after 28 games it had still not learned how to win an end game with two kings against one in a double corner.

Apparently, rote learning is of the greatest help either under conditions when the results of any specific action are long delayed or in those situations where highly specialized techniques are required. Contrasting with this, the generalization procedure is most helpful in situations in which the available permutations of conditions are large in number and when the consequences of any specific action are not long delayed.

Procedures Involving Both Forms of Learning

The next obvious step is to combine the better features of the rote-learning procedure with a generalization scheme. This must be done with some care, since it is not practical to update the previously saved information after every change in the evaluation polynomial. A compromise solution might be to save only a very limited amount of information during the early stages of learning and to increase the amount as warranted by the increasing stability of the evaluation coefficient with learning. For example, the program could be arranged to save only the piece-advantage term at the start. At some stage in the learning process the next term could be added, perhaps when no change had been made in the parameter used for this term during some fairly long period, say for three complete games. If and when the program is able to play an additional period without changes in the next parameter, this could also be added, et cetera. Whenever a change does occur in a parameter previously assumed to be stable, the entire memory tape could be reviewed, all terms involving the changed parameter and those lower on the list could be expunged, and the program could drop back to the earlier condition with respect to its term-saving schedule.

Another solution would be to utilize the generalization scheme alone until it had become fairly stable and to introduce rote learning at this time. It is, of course, perfectly feasible to salvage much of the learning which has been accumulated by both of the programs studied to date. This could be done by appending an abridged form of the present memory tape to the generalization scheme in its present stage of learning and by proceeding from there in accordance with the first solution proposed above.

Future Development

While it is believed that these tests have reached the stage of diminishing returns, some effort might well be expended in an attempt to get the program to generate its own parameters for the evaluation polynomial. Lacking a perfectly general procedure, it might still be possible to generate terms based on theories as proposed by students of the game. This procedure would be at variance with the writer's previous philosophy, but it is highly likely that similar compromises will have to be made when one attempts to apply learning procedures to problems of economic importance.

Conclusions

As a result of these experiments one can say with some certainty that it is now possible to devise learning schemes which will greatly outperform an average person and that such learning schemes may eventually be economically feasible as applied to real-life problems.

Appendix A: Programming Details

Approximate Size of Program

Basic checker-playing routine	1100 instructions
Input, move verification and output	1400 instructions
Game starting and terminating routines	600 instructions
Loaders, table generators, dumping, et cetera	850 instructions
Statistical and analytical routines	700 instructions
Rote-learning routines	1500 instructions
Generalization-learning routines	650 instructions
Tables and constants for basic play	700 words
Working space for basic play	2000 words
Working space for generalization learning	500 words
Working space for rote learning	Balance of memory

Approximate Computation Times

To find all available moves from given board position	2.6 milliseconds
To make a single move and find resulting board position	1.5 milliseconds
To evaluate a board position (4 terms)	2.4 milliseconds
To find score for a saved board position (rote learning)	2.3 milliseconds
To evaluate position (with 16 terms for generalization learning)	7.5 milliseconds

Board Representations

The standard checkerboard numbering system (see Appendix B) is used in communicating with the machine. A modified numbering system is used for internal computations, the numbers shown on the squares in Fig. 5 corresponding to the bit positions in an IBM 704 word. Any given board position is represented by four such words; one word (*FA*) containing 1's

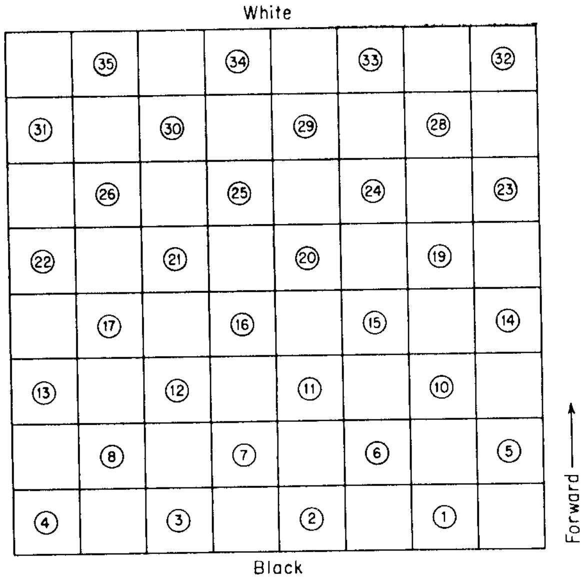


Figure 5. Checkerboard notation for internal computations.

in those bit positions corresponding to squares containing pieces of the color whose turn it is to move and which normally move in a forward direction. To be specific, if it is Black's turn to move (*i.e.*, if Black is "active") *FA* designates the location of all of Black's pieces, both men and kings. Conversely, if White is active, *FA* designates the location of White's kings only, since White's men can only move in the direction arbitrarily called *backward.* The other words designate, respectively: *BA,* backward active pieces; *FP,* forward passive pieces; and *BP,* backward passive pieces.

To conserve space when writing on tape, three words are used to record board positions with kings, and only two words are used for board positions without kings. These are saved in a standardized form, as explained in the text.

Possible moves are designated by five words; one word to indicate by its sign (with the word itself containing other information) whether the moves are jumps or not. (If a jump is available, only jump moves are saved.) The other four words designate the location of those pieces which can move in the four different diagonal directions: *RF,* for right forward; *LF,* for left forward; *LB* for left backward; and *RB,* for right backward, respectively.

By reference to Fig. 5, it will be observed that a right-forward move results in an increase of 4 in the square designation, while a left-forward move results in an increase of 5. Bit positions 9, 18 and 27 do not appear on the board. This notation makes it possible to compute available moves for all pieces simultaneously. Having previously computed a word called *EMPTY,* which contains 1's in locations corresponding to all unoccupied squares, one can compute *RF,* for the normal move case, in four instructions, as listed below (in IBM 704 symbolic language):

```
CLA     EMPTY     (puts word EMPTY into the accumulator)
ALS     4         (shifts word to left by 4 positions)
ANA     FA        (forms logical AND between EMPTY and FA)
STO     RF        (stores word as newly computed RF)
```

Jump moves are computed by a simple extension of this procedure. Multiple jumps are handled as a sequence of single jumps separated by null-reply moves.

Additional Timesaving Expedients

Bit counting is done by a table look-up procedure in a closed subroutine of 16 executed instructions (408 microseconds). This requires a 256-word table which is generated at the start by a 13-word program. Similar table look-up procedures are used, to turn a word end for end, and to locate the 1's in a word for move reporting.

Multiplications are usually avoided. In several places where multiplication by small integers must be done, it is programmed in terms of shifts and logical operations.

During the look-ahead procedure a complete record is kept of the sequence of board positions currently under investigation. As a result, no computing is needed to retract moves.

Appendix B: Sample Games from the Second Series with Generalization Learning

Typical Openings

The first eight moves of selected games in which Alpha played Black against Beta, showing the way in which different types of play were tried.

G-4	*G-6*	*G-12*	*G-17*	*G-19*	*G-21*	*G-31*	*G-37*	*G-39*	*G-41*	*G-43*
10 14	11 16	11 16	11 16	11 16	11 16	11 16	12 16	11 16	10 14	11 16
24 19	22 18	22 17	24 20	24 20	24 20	23 18	24 20	24 20	24 20	23 19
14 18	16 20	16 20	10 14	7 11	8 11	7 11	8 12	10 15	11 15	16 23
23 14	18 14	17 13	20 11	22 17	28 24	27 23	28 24	20 11	27 24	26 19
9 18	9 18	9 14	8 15	10 14	10 14	16 20	10 14	7 16	7 10	8 11
22 15	23 14	23 18	22 17	17 10	23 18	23 19	23 18	21 17	23 18	22 17
11 18	10 17	14 23	7 11	6 15	14 23	20 27	14 23	6 10	14 23	10 14
21 17	21 14	27 18	17 10	28 24	27 18	31 24	27 18	23 19	26 19	17 10

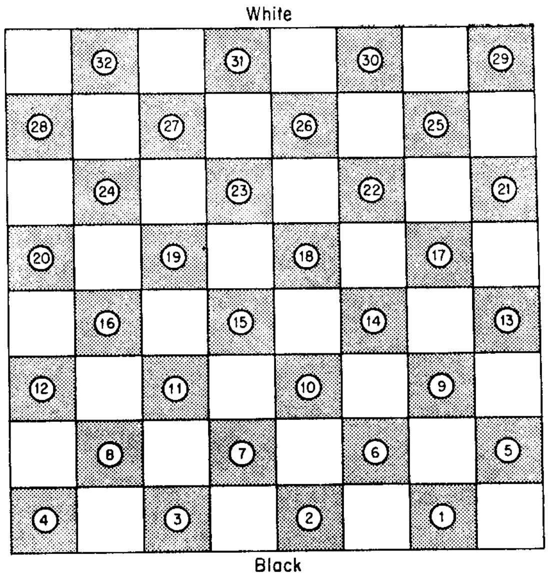


Figure 6. Square designations used in reporting games.

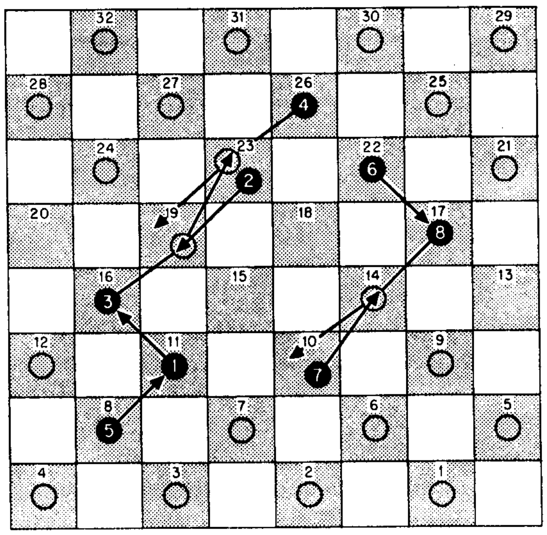


Figure 7. Eight-move opening utilizing generalization learning. (See Appendix B, Game *G-43*.)

Typical Games

Sample games in which Alpha played White against forced Beta openings.

G-1	*G-18*	*G-30*	*G-40*	*G-1*	*G-18*	*G-30*	*G-40*
12 16	12 16	12 16	10 14	9 13	12 16	9 14	4 8
24 19	24 20	24 20	24 20	1 6	24 20	18 9	1 6
8 12	8 12	8 12	11 15	13 17	16 19	8 11	10 14
22 18	28 24	28 24	27 24	32 27	29 25	15 8	6 10
10 14	10 15	10 14	7 10	16 20	13 17	4 11	14 17
26 22	22 18	22 18	23 18	18 14	10 7	19 15	10 15
16 20	15 22	6 10	14 23	11 15	2 11	11 18	17 21
30 26	25 18	24 19	26 19	6 10	14 10	23 14	32 28
11 16	7 10	1 6	10 14	15 18	19 23	13 17	5 9
28 24	18 14	32 28	19 10	14 9	21 14	9 5	27 24
7 11	10 17	3 8	6 15	Terminated	23 26	12 16	20 27
22 17	21 14	26 22	22 17	manually	10 7	28 24	19 16
3 8	9 18	9 13	2 7		26 30	17 22	12 19
17 10	23 14	18 9	17 10		25 21	6 10	15 22 31
6 15 22	6 9	5 14	7 14		30 26	30 25	9 14
26 17	30 25	22 18	24 19		7 3	1 6	31 26
9 13	9 18	6 9	15 24		11 15	25 21	14 18
17 14	26 23	25 22	28 19		14 10	5 1	28 24
2 7	3 8	2 6	14 17		5 9	21 17	8 11
23 18	23 14	30 25	21 14		10 6	24 20	24 19
16 23	1 6	14 17	9 18		15 19	16 19	21 25
14 10	27 23	21 14 5	25 22		6 1	20 16	30 21
7 14	6 9	6 9	18 25		26 22	17 13	Beta
18 9	14 10	18 15	29 22		1 6	6 2	concedes
5 14	9 13	11 18	5 9		9 13	13 17	
27 18 9	25 21	20 11 2	31 27		20 16	10 6	
20 27	11 15	10 14	1 5		19 23	Beta	
31 24	20 11	22 15	20 16		6 9	concedes	
12 16	15 18	14 17	3 7		23 27		
21 17	23 14	5 1	22 17		16 11		
13 22	8 15	17 21	8 11		22 25		
25 18	24 19	25 22	17 13		11 7		
1 5	15 24	21 25	11 20		25 30		
9 6	32 28	22 18	13 6		7 2		
5 9	24 27	25 30	7 10		27 32		
6 1	31 24	2 6	6 1		70 Move termination		

Appendix C: Evaluation Polynomial Details for Second Series

Method of Computing Terms

The 16 terms called for in the evaluation polynomial are computed, individually, by taking the value of the appropriate parameter, as defined below, for the board position under consideration and subtracting the value of this same parameter computed for the board position just prior to the last move (with the necessary reversal in the definitions of active and passive sides). This difference is then multiplied by the corresponding program-computed coefficient, which can vary between -2^{18} and $+2^{18}$, and credited to the side which was passive on the board position under consideration.

Definitions of Parameters

ADV (Advancement)
The parameter is credited with 1 for each passive man in the 5th and 6th rows (counting in passive's direction) and debited with 1 for each passive man in the 3rd and 4th rows.

APEX (Apex)
The parameter is debited with 1 if there are no kings on the board, if either square 7 or 26 is occupied by an active man, and if neither of these squares is occupied by a passive man.

BACK (Back Row Bridge)
The parameter is credited with 1 if there are no active kings on the board and if the two bridge squares (1 and 3, or 30 and 32) in the back row are occupied by passive pieces.

CENT (Center Control I)
The parameter is credited with 1 for each of the following squares: 11, 12, 15, 16, 20, 21, 24 and 25 which is occupied by a passive man.

CNTR (Center Control II)
The parameter is credited with 1 for each of the following squares: 11, 12, 15, 16, 20, 21, 24 and 25 that is either currently occupied by an active piece or to which an active piece can move.

CORN (Double-corner Credit)
The parameter is credited with 1 if the material credit value for the active side is 6 or less, if the passive side is ahead in material credit, and if the active side can move into one of the double-corner squares.

CRAMP (Cramp)
The parameter is credited with 2 if the passive side occupies the cramping square (13 for Black, and 20 for White) and at least one other nearby square (9 or 14 for Black, and 19 or 20 for White), while certain squares (17, 21, 22 and 25 for Black, and 8, 11, 12 and 16 for White) are all occupied by the active side.

DENY (Denial of Occupancy)
The parameter is credited with 1 for each square defined in MOB if on the next move a piece occupying this square could be captured without an exchange.

DIA (Double Diagonal File)
The parameter is credited with 1 for each passive piece located in the diagonal files terminating in the double-corner squares.

DIAV (Diagonal Moment Value)
The parameter is credited with ½ for each passive piece located on squares 2 removed from the double-corner diagonal files, with 1 for each passive piece located on squares 1 removed from the double-corner files and with 3/2 for each passive piece in the double-corner files.

DYKE (Dyke)
The parameter is credited with 1 for each string of passive pieces that occupy three adjacent diagonal squares.

EXCH (Exchange)
The parameter is credited with 1 for each square to which the active side may advance a piece and, in so doing, force an exchange.

EXPOS (Exposure)
The parameter is credited with 1 for each passive piece that is flanked along one or the other diagonal by two empty squares.

FORK (Threat of Fork)
The parameter is credited with 1 for each situation in which passive pieces occupy two adjacent squares in one row and in which there are three empty squares so disposed that the active side could, by occupying one of them, threaten a sure capture of one or the other of the two pieces.

GAP (Gap)
The parameter is credited with 1 for each single empty square that separates two passive pieces along a diagonal, or that separates a passive piece from the edge of the board.

GUARD (Back-row Control)
The parameter is credited with 1 if there are no active kings and if either the Bridge or the Triangle of Oreo is occupied by passive pieces.

HOLE (Hole)
The parameter is credited with 1 for each empty square that is surrounded by three or more passive pieces.

KCENT (King Center Control)
The parameter is credited with 1 for each of the following squares: 11, 12, 15, 16, 20, 21, 24 and 25 which is occupied by a passive king.

MOB (Total Mobility)
The parameter is credited with 1 for each square to which the active side could move one or more pieces in the normal fashion, disregarding the fact that jump moves may or may not be available.

MOBIL (Undenied Mobility)
The parameter is credited with the difference between MOB and DENY.

MOVE (Move)
The parameter is credited with 1 if pieces are even with a total piece count (2 for men, and 3 for kings) of less than 24, and if an odd number of pieces are in the move system, defined as those vertical files starting with squares 1, 2, 3 and 4.

NODE (Node)
The parameter is credited with 1 for each passive piece that is surrounded by at least three empty squares.

OREO (Triangle of Oreo)
The parameter is credited with 1 if there are no passive kings and if the Triangle of Oreo (squares 2, 3 and 7 for Black, and squares 26, 30 and 31 for White) is occupied by passive pieces.

POLE (Pole)
The parameter is credited with 1 for each passive man that is completely surrounded by empty squares.

RECAP (Recapture)
This parameter is identical with Exchange, as defined above. (It was introduced to test the effects produced by the random times at which parameters are introduced and deleted from the evaluation polynomial.)

THRET (Threat)
The parameter is credited with 1 for each square to which an active piece may be moved and in so doing threaten the capture of a passive piece on a subsequent move.

Binary Connective Terms

The abbreviations used for the terms of this type which have been employed are listed below, in the order of $A \cdot B$, $A \cdot \bar{B}$ $\bar{A} \cdot B$, and $\bar{A} \cdot \bar{B}$, where A and B are the two respective parameters heading the sublists of abbreviations.

Denial of occupancy—total mobility	Undenied mobility—denial of occupancy	Undenied mobility—center control I
DEMO	MODE 1	MOC 1
DEMMO	MODE 2	MOC 3
DDEMO	MODE 3	MOC 2
DDMM	MODE 4	MOC 4

Evaluation Polynomial (First 12 Terms Only) after 42 Games, during Which a Total of 1,039 Different Sets of Adjustments Were Made to the Terms and Their Coefficients[13]

Term	Correlation coefficient	Sign of coefficient	Power of 2 used as coefficient	Times adjusted
MOC 2	0.45	−	18	84
KCENT	0.40	+	16	127
MOC 4	0.35	−	14	95
MODE 3	0.33	−	13	210
DEMMO	0.27	−	11	132
MOVE	0.19	+	8	91
ADV	0.19	−	8	739
MODE 2	0.19	−	8	55
BACK	0.14	−	6	6
CNTR	0.13	+	5	12
THRET	0.13	+	5	442
MOC 3	0.10	+	4	89

Discarded Terms during 42 Games[13]

Term	Times adjusted before discard	Term	Times adjusted before discard
CORN	0	MODE 1	1
CRAMP	0	CENT	386
GUARD	0	MODE 4	0
EXPOS	162	FORK	400
DDMM	19	MOBIL	707
DYKE	115	POLE	11
MOC 1	1	HOLE	598
EXCH	445	GAP	792
DDEMO	53	MOB	608

[13] An additional 20 games have recently been played. Although some significant changes were noted, the general stabilization of the learning process suggested by Fig. 4 has been confirmed. During this play, 412 more adjustments were made to the terms and their coefficients and 12 additions were made to the list of discarded terms.

Appendix D: Game Played by Mr. R. W. Nealey and the Samuel Checker Program

In the summer of 1962, at the request of the editors of this collection, Dr. Samuel arranged a match between his checker-playing program (on an IBM 7090 computer) and a human checker champion.

Mr. Robert W. Nealey is described in the *IBM Research News* for August, 1962, as "a former Connecticut checkers champion, and one of the nation's foremost players."

The Samuel program bested Mr. Nealey in the game reprinted below. The annotations were made by Dr. Samuel. Mr. Nealey's comments, as quoted by the *IBM Research News*, are as follows:

Our game . . . did have its points. Up to the 31st move, all of our play had been previously published, except where I evaded "the book" several times in a vain effort to throw the computer's timing off. At the 32-27 loser and onwards, all the play is original with us, so far as I have been able to find. It is very interesting to me to note that the computer had to make several star moves in order to get the win, and that I had several opportunities to draw otherwise. That is why I kept the game going. The machine, therefore, played a perfect ending without one misstep. In the matter of the end game, I have not had such competition from any human being since 1954, when I lost my last game.

Nealey (WHITE) vs. Samuel Checker Program (BLACK)

Date: July 12, 1962
Place: Yorktown, New York
Mr. Nealey was given the option and chose to defend. The Old Fourteenth opening was followed.

11	15		
23	19		
8	11		
22	17		
4	8		
17	13		25-22 would restrict Black's variety of play a little more.
15	18		
24	20		Lee's Old Fourteenth, Var. 9. 11-15 is the trunk move.
9	14		
26	23		Doran's Var. 100 listed as an even game.
10	15		
19	10		
6	15		
28	24		Doran lists 23-19 as giving an easier game for White.
15	19		An aggressive move for Black.
24	15		
5	9		
13	6		
1	19	26	
31	22	15	
11	18		Still in Lee's Var. 9 and Doran's Var. 100.

30	26	This is probably a poor move on Mr. Nealey's part.
8	11	A good reply maintaining control of the center.
25	22	
18	25	
29	22	
11	15	
27	23	
15	19	
23	16	
12	19	
32	27	White makes a losing move.
19	24	The obvious reply, guaranteeing Black a king.
27	23	
24	27	
22	18	
27	31	Black now has his king.
18	9	
31	22	
9	5	
22	26	A delaying move to force White to advance.
23	19	
26	22	
19	16	
22	18	
21	17	
18	23	
17	13	
2	6	Le coup de maitre. A Black win is now certain.
16	11	
7	16	
20	11	
23	19	Le coup mortel

White concedes.

Location of Black pieces-3,6,19K
Location of White pieces-5,11,13.

section 3

Machines That Prove Mathematical Theorems

The discovery of proofs for mathematical theorems constitutes intellectual activity of a high order. The learning of mathematical proof techniques is considered by many to be good training in general problem-solving discipline.

Ironically, the elegant proofs that mathematicians present in their scholarly reports and textbooks usually do not provide one with much insight into the actual mental processes of discovery that were used to find the proofs. Occasionally one catches a glimpse of these processes during a classroom lecture by an excellent teacher of mathematics. Such an experience is probably the closest point of contact with the private problem-solving world of the mathematician.

The fascination with mechanical theorem proving for most of the researchers working in this area lies less with the end (the production of theorems, perhaps new and important) than with the means (a thorough understanding of the organization of information processing activity in mathematical discovery). It is felt that understanding these problem-solving processes is an important step toward the programming of more complex and general problem-solving processes for a variety of intellectual tasks. In theorem-proving research, as in the game-playing studies, the simplicity of the formal system allows most of the research effort to be devoted to understanding problem-solving processes rather than to modeling the task environment.

Not all work on mechanical theorem proving is concerned with problem-solving means—the more general problem. Some researchers, notably Wang, are deeply concerned with the end—the production of theorems. They have achieved impressive results using advanced and sophisticated mathematical decision rules.

The Logic Theorist is a computer program which discovers proofs to the theorems in symbolic logic (chapter 2 of the Whitehead-Russell *Principia Mathematica*). It uses proof methods no more advanced than those available to the student just beginning a first course in *Principia.*

The Logic Theorist was programmed by Newell, Shaw, and Simon in early 1956. It was the first heuristic program fully realized on a computer, the first foray by artificial intelligence research into high-order intellectual processes.

It is interesting to note that the Logic Theorist was accompanied by, or gave rise to, another development of great importance to artificial intelligence research and the computer sciences in general: the first list processing computer language. The language was reported by Newell and Shaw in a companion piece to the paper which is reprinted (the piece was called "Programming the Logic Theory Machine," and the language is called Information Processing Language, or IPL).

The work of Gelernter and his associates extends heuristic programming ideas to the proof of theorems in euclidean geometry. His program makes use of heuristic methods where they are most effective, but it also applies more powerful, more direct symbol manipulation processes where these are useful. Of special interest in the geometry proof program is the use of the diagram as a heuristic device in guiding search of the subproblem structure.

The geometry group, too, developed a list processing language for writing their program. Called FORTRAN List Processing Language (FLPL), it combines the ordinary capabilities of FORTRAN (for specifying numerical computations) with certain list processing features.

H. Gelernter and J. R. Hansen are members of the research staff of the IBM Research Laboratory in Yorktown Heights, N.Y. D. W. Loveland is at the Courant Institute, New York, N.Y.

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, ot 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.

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.

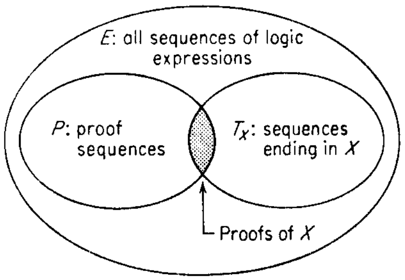

E: all sequences of logic expressions
P: proof sequences
T_X: sequences ending in X
Proofs of X

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

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.

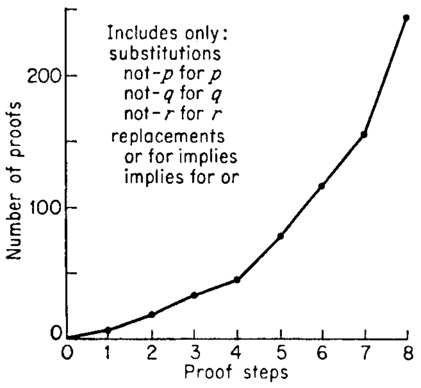


Figure 2. Number of proofs generated by first few steps of British Museum algorithm.

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-

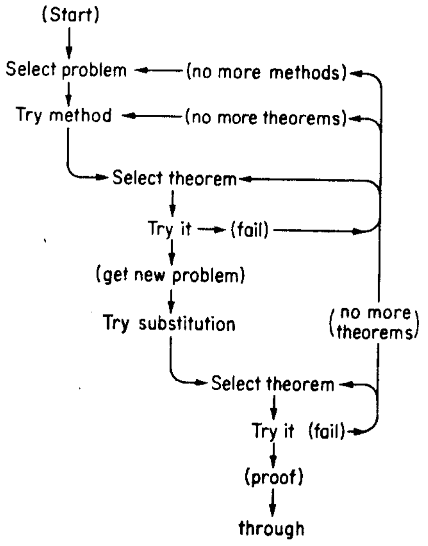


Figure 3. General flow diagram of LT.

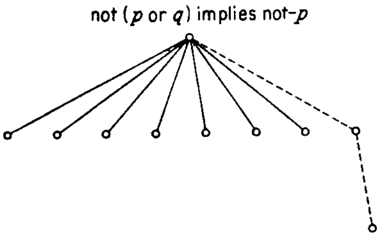


Figure 4. Subproblem tree of proof by LT of (2.45) (all previous theorems available).

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-

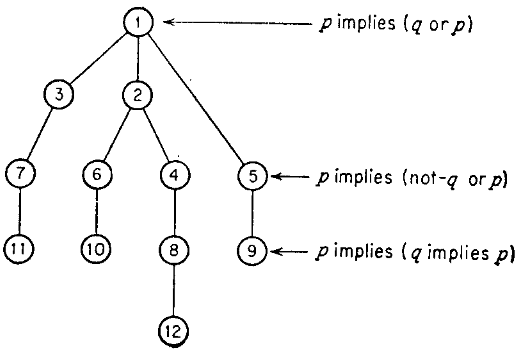


Figure 5. Proof tree of proof 2.02 by British Museum algorithm (using axiom 1.3).

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 modification 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 detachment. 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

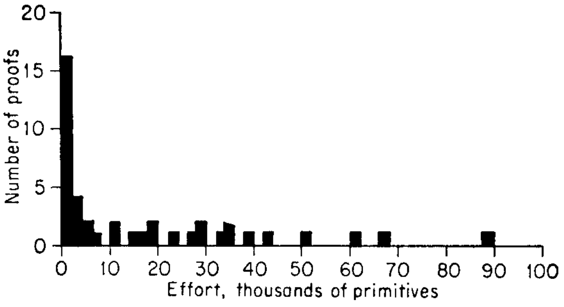


Figure 6. Distribution of LT's proofs by effort. Data include all proofs from attempts on the first 52 theorems in chap. 2 of *Principia.*

—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

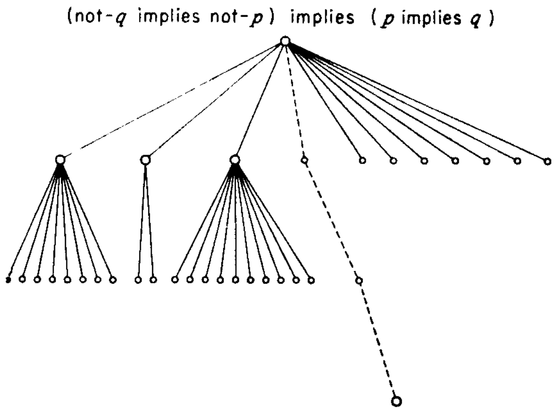


Figure 7. Subproblem tree of proof by LT of (2.17) (all previous theorems available).

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

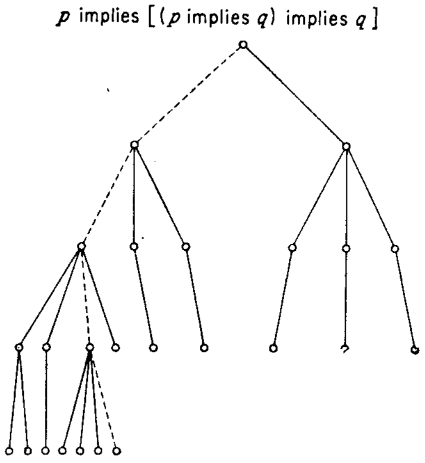


Figure 8. Subproblem tree of proof by LT of (2.27) (using the axioms).

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.

REALIZATION OF A GEOMETRY-THEOREM PROVING MACHINE

by H. Gelernter

Introduction

Few of those who have seen a modern high-speed digital computer digest and transform a mass of data in less time than it takes to follow the process in the mind can suppress a certain amount of speculation concerning the future of such machines. Under the assumption that the computer is operating at the mere threshhold of its capacity in performing the tasks we have thus far delegated to it, a long-range program directed at the problem of "intelligent" behavior and learning in machines has been established at the IBM Research Center in New York (Gelernter and Rochester, 1958). In particular the technique of heuristic programming is under detailed investigation as a means to the end of applying large-scale digital computers to the solution of a difficult class of problems currently considered to be beyond their capabilities; namely those problems that seem to require the agent of human intelligence and ingenuity for their solution. It is difficult to characterize such problems further, except, perhaps, to remark rather vaguely that they generally involve complex decision processes in a potentially infinite and uncontrollable environment.

If, however, we should restrict the universe of problems to those that amount to the discovery of a proof for a theorem in some well-defined formal system, then the distinguishing characteristics of those problems of special interest to us are brought clearly into focus. We should like our machine to be able to prove many of the theorems presented to it in a formal system that is manifestly undecidable. Further, as the machine gains "experience" in proving theorems, we should expect it to be able to solve problems that were earlier beyond its capabilities.

The requirement that a machine should deal with undecidable systems places a fundamental restriction on its modus operandi. Finding a suitable algorithm, the obvious technique for the solution of problems on a digital computer, is no longer acceptable for the simple reason that no such algorithm exists. An exhaustive search for the initial axioms and theorems of the proof, combined with exhaustive development of the proof sequence by systematically applying the rules of transformation until the required proof has been produced, has been shown to be much too time-consuming for so simple a logic as propositional calculus (Newell, Shaw and Simon, 1957*a*). It is a fortiori out of the question for any of the more interesting logics. A remaining alternative is to have the machine rely upon heuristic methods, as people usually do under similar circumstances.

Heuristic Methods

A heuristic method is a provisional and plausible procedure whose purpose is to discover the solution of a particular problem at hand. The use of heuristic methods by the human mathematician is quite well understood, at least in its less subtle forms. The reader is referred to the excellent two-volume treatise by Prof. G. Polya (1954) for a definitive treatment of heuristics and mathematical discovery. A machine that functioned under the full set of principles indicated by Polya would be a formidable problem-solver in mathematics, and would be well on the way toward satisfying Turing's requirements for a machine able to compete successfully in the "imitation game" (1950). Such a machine, however, lies in the indefinite future, for the art of instructing a computer is yet in too primitive a state to consider translating Polya into machine language. As a representative problem more in keeping with the present state of computer technology, we have selected the discovery of proofs for theorems in elementary euclidean plane geometry in the manner, let us say, of a high-school sophomore. This problem contains in relatively pure form the difficulties we must surmount in order to attain our stated goal. It must be emphasized that although plane geometry will yield to a decision algorithm, the proofs offered by the machine will not be of this nature. The methods developed will be no less valid for problem-solving in systems where no such decision algorithm exists.

Although we have narrowed the scope of our study to include only those machines that deal with formal systems, there is ample justification for such a restriction. First, the concept of a problem is now well defined, as is the concept of a solution for that problem. Second, our ultimate goal stands clearly before us; it is the design of an efficient theorem-prover in some undecidable system. And, finally, just as manipulation of numbers in arithmetic is the fundamental mode of operation in contemporary computers, manipulation of symbols in formal systems is likely to be the fundamental operating mode of the more sophisticated problem-solving computers of the future. It seems clear that while the problems of greatest concern to lay society will be, for the most part, not completely formalizable, they will have to be expressed in some sort of formal system before they can be dealt with by machine.[1]

Our problem, then, is a statement (or string) in some formal logistic system. A solution for the problem will be a sequence of statements, each of which comprises a string of symbols of the alphabet of the system. The last string of the solution will be the problem itself; the first will always be an axiom or previously established theorem of the system. Every other string will be immediately inferable from some set preceding it or will itself be an axiom or previously established theorem.[2] It is the task of the machine to choose from its stock of axioms and theorems the appropriate ones for the base of the proof, and to generate from these the remaining strings necessary to complete the proof.

The problem of theorem-proving is, in a sense, of a particularly simple nature. Once a sequence of expressions is found that passes the test for a proof of the theorem (such a test always exists), one may, so to speak, "close the book" on that problem, provided that no stipulations have been made concerning the elegance required of the proof. But, basing our estimate on the work of Newell, Shaw, and Simon (1957), any computer extant would require times of the order of a thousand years to prove a not uncommon ten-step geometry theorem by exhaustively developing sequences until one emerged that passed the test for a proof. What is clearly called for is a technique for generating sequences with a much higher *a priori* probability of being the solution to the problem than those generated by an exhaustion algorithm.

As did the *Logic Theorist* of Newell, Shaw, and Simon, the geometry machine relies upon the well-known analytic method to achieve this end. By working backward, the machine is assured that every sequence it considers does indeed terminate in the required theorem. This in itself, however, represents no striking improvement over exhaustion without additional heuristics, for the advantages of working backward are purchased at a steep price; each sequence generated, while terminating properly, is no longer guaranteed to be a proof of anything at all. Indeed, most of the strings generated in this way will be false! But it is here that the great

[1] For a critique of some attempts to formalize scientific, but nonmathematical theories, see Dunham, Fridshal, and Sward (1959).

[2] The machine will use the deduction theorem to get $\vdash\{\text{premises}\}\supset\{\text{conclusions}\}$ from $\{\text{premises}\}\vdash\{\text{conclusions}\}$.

power of the analytic method lies, for if one could find a way of making their falseness manifest, such sequences could be immediately rejected, allowing most of the deadwood to be pruned away from the highly branched problem-solving tree. The set of sequences generated under such a process would contain fewer members by many orders of magnitude by the time the search reached any depth, and the density of possible proofs for the theorem among them would be proportionately greater. It is here, too, that the geometry machine finds the additional theorem-proving power necessary for the complex formal system assigned to it; theorem-proving power that was not necessary, and therefore not sought for in the propositional calculus machine of Newell, Shaw, and Simon (Polya, 1954). Like the human mathematician, the geometry machine makes use of the potent heuristic properties of a diagram to help it distinguish the true from the false sequences. Although the diagram is useful to the machine in other ways as well, the single heuristic "Reject as false any statement that is not valid in the diagram" is sufficient to enable the machine to prove a large class of interesting theorems, some of which contain a certain trivial kind of construction.

Before examining the internal structure of the geometry machine in some detail, we remark on two fundamental, if obvious, principles that must guide the choice of heuristics for any problem-solving machine. A heuristic is, in a very real sense, a filter that is interposed between the solution generator and the solution evaluator for a given class of problems. The first requirement for such a filter is a consequence of the fact that its introduction into the system is never costless. It must, therefore, be sufficiently "nonporous" to result in a net gain in problem-solving efficiency. Secondly, a heuristic will generally remove from consideration a certain number of sequences that are quick and elegant solutions, if not indeed all solutions, to some potential problems within the domain of the problem-solving machine. The filter must, then, be carefully matched to that subclass of problems in the domain containing those that are considered "interesting," and are therefore likely to be posed to the machine. For a given class of heuristics, the balance between these essentially opposing requirements is largely a function of the organization and computing power of the machine, and can under certain rather easily attainable conditions be quite critical. In the case of the *Logic Theorist*[3] experiments with varying "strengths" of a particular heuristic (the similarity test) indicated that the optimum porosity of that heuristic varied markedly with the length of the

[3] The designers of the *Logic Theorist* were not unaware of this heuristic device. In a later version of that machine, they did, in fact, include some syntactic heuristics to reject false subgoals. To use a semantic interpretation of the propositional calculus (a truth table, for example) for this purpose would have reduced the *Logic Theorist* to triviality.

problem and the number of theorems already established in the theorem memory, a consequence of the limited storage capacity of the computer.

The Geometry Machine

With the object of our research program clearly determined, there were a number of specific alternatives to theorem-proving in Euclidean geometry that might have been adopted as a test problem; the evaluation of indefinite integrals, for example, or theorem-proving in the pure functional calculus. The decisive point in favor of geometry was the great heuristic value of the diagram. The creative scientist generally finds his most valuable insights into a problem by considering a model of the formal system in which the problem is couched. In the case of Euclidean geometry, the semantic interpretation is so useful that virtually no one would attempt the proof of a theorem in that system without first drawing a diagram; if not physically, then in the mind's eye. If a calculated effort is made to avoid spurious coincidences in the figure, one is usually safe in generalizing any statement in the formal system that correctly describes the diagram, with the notable exception of those statements concerning inequalities. Further geometry provides illustrative material in treatises and experiments in human problem-solving. It was felt that we could exchange valuable insights with behavioral scientists during the course of our research. In any event, elementary Euclidean geometry is comprehensible to every segment of the scientific community to which we should wish to communicate our results. Finally, it should not be a difficult task to generalize our machine to include the more interesting case of the non-Euclidean geometrics. A program of the same theorem-proving power as our Euclidean theorem-prover should be sufficient to prove a large class of non-obvious theorems in non-Euclidean geometry. A machine furnished with a non-Euclidean diagram (no more difficult to supply than the Euclidean one in suitable analytic form) encounters none of the assault on rationality experienced by a human mathematician searching from some heuristic insight into a theorem by considering a non-Euclidean diagram.

The formalization of geometry must be carried out within the framework of the lower functional calculus. Since we are interested in having the machine produce proofs comparable to those of a high-school student, we have preferred to construct a more or less *ad hoc* system following the scheme of most elementary texts, rather than to adopt as a primitive basis the fundamental axiomatization of Tarski, Hilbert, or Forder. No attempt has been made to provide a formalization that is either complete or non-redundant. If at some later time, the machine is able to prove one axiom from the others, that axiom will be discarded and we shall applaud the elegance displayed by our automaton. With regard to completeness, the

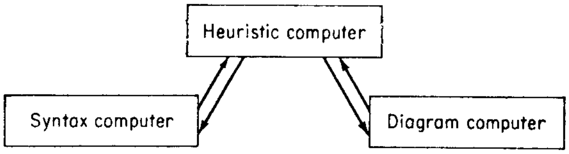


Figure 1.

machine is granted the same privileges enjoyed by the high-school student who is always assuming (*i.e.*, introducing as additional axioms) the truth of a plethora of "obviously self-evident" statements concerning, for example, the ordering properties of points on a line and the intersection properties of lines in a plane. Some of these statements are indeed independent of his original axioms, and must be introduced to complete the system. Most could be derived (but usually with some difficulty) from what he already has. There is nothing essentially wrong with this procedure of extracting assumptions from the model, provided that one is fully aware that this is being done (of course, this is rarely the case for the average student), and it simplifies the proof considerably without invalidating it. The geometry machine explicitly records its assumptions for a given proof. It could, if necessary, minimize the danger that it is proving a specific instance of a given theorem by drawing alternate diagrams to test the generality of its assumptions.

The geometry machine is in reality a particular state configuration of the IBM 704 electronic Data Processing Machine specified by a rather long and complex program written for the computer. Its organization falls naturally into three parts: a *syntax computer* and a *diagram computer* both embedded in an executive routine, which is a *heuristic computer*. The flow of control is indicated in Fig. 1.

Manipulation of the formal system is relegated to the syntax computer, which has within it the equivalent of most of the syntactic heuristics used by the *Logic Theorist*.[4] The diagram computer contains a coordinate representation of the theorem to be established together with a series of routines that produce a qualitative description of the diagram. It is important to point out that although the procedures of analytic geometry are used to generate the description, the only information transmitted to the heuristic computer (there is no direct link between the diagram and the formal system) is of the form: "Segment AB appears to be equal to segment CD in the diagram," or "Triangle ABC does not contain a right angle in the diagram." The behavior of the system would not be changed if the diagram computer were replaced by a device that could draw figures on paper and scan them.

[4] The process of *chaining* as defined by Newell et al. is under the control of the heuristic computer.

The major function of the heuristic computer for our first system, the subject of this report, is to compare strings generated by the syntax computer (working backward) with their interpretation in the diagram, rejecting those sequences that are not supported by the model. In addition to the above, the heuristic computer performs several other tasks. Among these are the organization of the proof-search process and the recognition of the syntactic symmetry of certain classes of strings. The latter function produces behavior equivalent to that of the human mathematician who, when A and B are syntactically symmetric, and both must be established, will merely prove A, and say "Similarly, B." It is an important feature, and is described in detail in an earlier report (Gelernter, 1959*a*). The procedures above are clearly independent of geometry; they are applicable to any formal system with its corresponding interpretation. The heuristic computer applies some additional semantic heuristics that are not indepedent of geometry. These may be "switched off" so that the behavior of the machine can be observed with and without specific geometry heuristics.

The character of the theorem-proving machine is determined largely by the heuristic computer. Modifications and improvements in the system (the introduction of learning processes, for example) will be made by modifying this part of the program.

Our first system does not "draw" its own initial figure, but is, instead, supplied with the diagram in the form of a list of possible coordinates for the points named in the theorem. This point list is accompanied by another list specifying the points joined by segments. Coordinates are chosen to

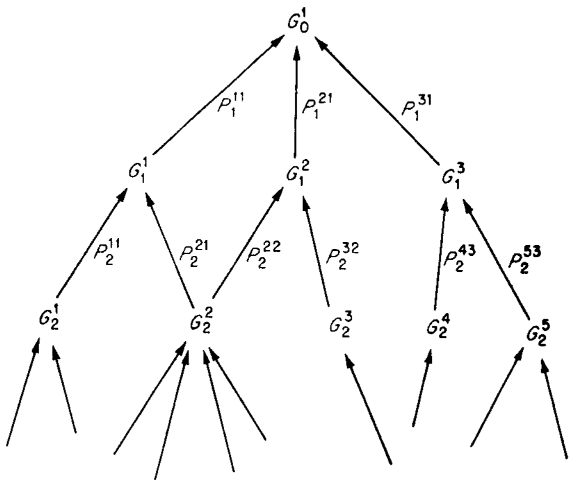


Figure 2. Problem-solving graph. The nodes G_i^α represent subgoals of order i, with α numbering the subgoals of a given order. $P_i^{\alpha\beta}$ is a transformation on G_i^α into G_{i-1}^β.

reflect the greatest possible generality in the figures. Later systems will construct their own interpretation of the premises, but since most problems for high school students are accompanied by a diagram, it was felt that we could dispense with this additional spate of programming at the current stage. When the machine is drawing its own figures, points will be chosen at random, subject to the constraints of the premises.

In working backward, the system generates a problem-solving graph, defined in the following way: Let G_0 be the formal statement to be established by the proof. It will be called the problem goal. If G_i is a formal statement with the property that G_{i-1} may be immediately inferred from G_i, then G_i is said to be a subgoal of order i for the problem. All G_j such that $j < i$ are higher subgoals than G_i, where G_0 is considered to be a subgoal of order zero. The problem-solving graph (Fig. 2) has as nodes the G_i, with each G_i joined to at least one G_{i-1} by a directed link. Each link represents a given transformation from G_i to G_{i-1}. The problem is solved when any G_i can be immediately inferred from the premises and axioms. If, as is generally the case in geometry, a given subgoal is a conjunction of statements, the graph splits at that point, and each parallel subgoal must be separately established. At any given time, the problem-solving graph is a complete representation of the current status of the proof-search process.

The organization of the heuristic computer (which is also the organization of the entire system) is displayed in greatly simplified form in Fig. 3. The diagram and syntax computers are accessible as subroutines to the heuristic computer. In operation, the machine executes the following processes (numbered below to correspond with like-numbered blocks in the flow chart).

1. The diagram is scanned to construct three lists, one containing every segment in the figure, one the angles, and one the triangles. Each element on a list is followed by a sublist describing that element.
2. The initial configuration of the system is set up, with the premises placed on a list of established formulas, and the conclusion on the problem-solving graph as a zero-order subgoal.
3. Definitions of nonprimitive predicates in the premises are added to the list of established formulae.
4. A subgoal to be established (the generating subgoal) is chosen from the problem-solving graph.
5. Appropriate axioms and theorems are selected from the theorem memory and, by working backward, a set of lower subgoals is generated such that if any one of these is established, the generating subgoal may be established by modus ponens and the generating axiom (or theorem). If the generating subgoal was labeled "provisionally fruitless" (see step 8), constructions are attempted (see below, p. 144).

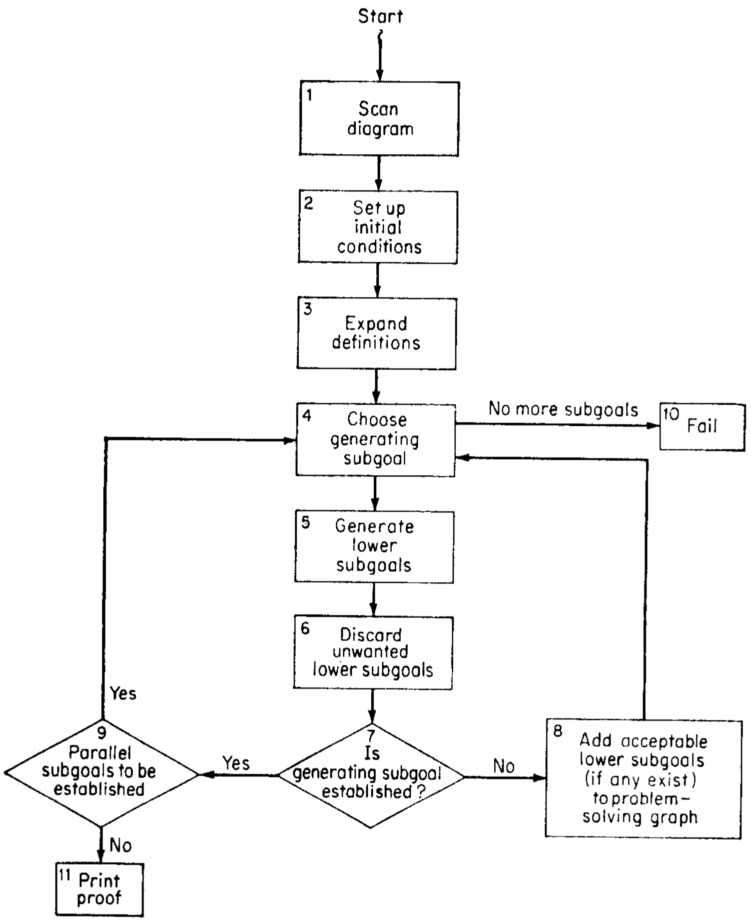

Start
1 Scan diagram
2 Set up initial conditions
3 Expand definitions
4 Choose generating subgoal
No more subgoals
10 Fail
5 Generate lower subgoals
6 Discard unwanted lower subgoals
7 Is generating subgoal established ?
Yes
No
8 Add acceptable lower subgoals (if any exist) to problem-solving graph
9 Parallel subgoals to be established
Yes
No
11 Print proof

Figure 3. Simplified flow chart for the geometry-theorem proving machine.

6. Subgoals that are not valid in the diagram are rejected, as are those that appear as higher subgoals on the graph (or are syntactically symmetric to some higher subgoal).

7. If any lower subgoal is valid by virtue of its instance on the list of established formulas or if it may be assumed from the diagram, the generating subgoal is established; otherwise—

8. Acceptable nonredundant lower subgoals are added to the graph, and a new subgoal generator is chosen (4). If there are no acceptable lower subgoals and a construction is possible at this point, the generating subgoal is designated as provisionally fruitless. If a construction is not possible, or if the machine has tried and failed to find one, the generating subgoal is designated as fruitless.

9. If the generating subgoal is established, it is added to the list of established formulas, together with all of its higher consequences as determined by the graph. If there are no parallel subgoals remaining to be established, the machine reconstructs the proof from the problem-solving graph and prints it (11).

10. If at any time, every free subgoal on the graph is fruitless, the machine fails, providing it has not previously exhausted its available storage or the patience of the operator.

It is within blocks 4, 5 and 6, where subgoals are chosen, developed, and discarded, that the major heuristics reside. These subprograms represent, if you like, the seat of our artificial intelligence.

Some Early Results

The geometry machine is able to prove many of the theorems within the scope of its ad hoc formal system using the diagram only to indicate which subgoals are probably valid. In this way, the following theorem is proved in less than a minute.[5]

Theorem: A point on the bisector of an angle is equidistant from the sides of the angle (see Fig. 4 in Appendix A).

In less than five minutes, the machine is able to find the attached proof, which requires the construction of an auxiliary segment.

Theorem: In a quadrilateral with one pair of opposite sides equal and parallel, the other pair of sides are equal (see Fig. 5 in Appendix B).

Although the introduction of a new element by the machine is impressive, the construction in this proof is essentially trivial, for the new segment merely joins two already existing points. It was discovered by the following process. In attempting to develop subgoals for the string "AB = CD," the machine could find none that were valid in the diagram. The normal procedure at this point is to seek an alternative path on the problem-solving graph. But when none is available (as was the case here, since the offending string is a zero-order subgoal), the machine reexamines those of the previously rejected subgoals containing instances of predicates for which there was no representation in the diagram. The machine then considers for each one an augmented set of premises such that the

[5] In the proofs displayed herein, the nonobvious predicates have the following interpretations:

OPP-SIDE XYUV	Points X and Y are on opposite sides of the line through points U and V.
SAME-SIDE XYUV	Points X and Y are on the same side of the line through points U and V.
PRECEDES XYZ	Points X, Y, and Z are collinear in that order.
COLLINEAR XYZ	Points X, Y, and Z are collinear.

interpretation does contain a representation of the predicate. If the string is valid in the augmented system, and there exist theorems permitting the required additional premises to be derived from the original set, then the string becomes a subgoal in the augmented system. The added premises specify a construction in the diagram that is permitted by virtue of the theorems through which they were derived. Returning to our example, the subgoal "$\Delta ABD \cong \Delta CDB$" is generated for the string "AB = CD," but the required triangles are not represented in the diagram until the premise "Segment BD exists" is added. The axiom "Two distinct points determine a segment" justifies the construction.[6] The entire process is a variant of the major heuristic above, and is clearly independent of the particular formal system under consideration. Note, too, that the process is finite, since no new points are introduced into the predicates; the old ones are merely reconsidered.

Our second example illustrates one further point. Although it is clear in the diagram (Fig. 5) that the transversal BD makes alternate interior angles with sides BC and AD, this is a consequence of the theorem "Opposite vertices of a convex quadrilateral fall on opposite sides of the diagonal through the other vertices." That this is not true of a general quadrilateral becomes clear when one considers the outside diagonal of a reflex quadrilateral. A completely rigorous solution, then, requires that one prove the lemma above if it is not already available, and that one demonstrate that the quadrilateral ABCD can only be convex. Rather than do this, the machine makes the usual assumption that the diagonal forms alternate interior angles with the opposite sides of the quadrilateral. Unlike the usual high-school text, however, the assumption is made explicit in the proof.

The theorem-proving system described thus far is adequate for many problems of greater complexity than the ones cited above. However, with a linear increase in the number of individual points mentioned in the premises, the rate of growth of the problem-solving graph increases exponentially and the time required to explore the graph increases correspondingly. If the machine were able to select those among a given set of subgoals that were more likely to lead to a solution, much of the wasted search time could be eliminated. Two specific geometry heuristics have been introduced to enable the machine to do this. The first is a routine that recognizes certain of the subgoals that are usually established in just one step. Identities are in this category, for example, as are equalities between angles that are observed to be vertical angles in the diagram. Such subgoals

[6] Our ad hoc formal system requires that the segments joining the vertices of a triangle be specified, as well as the vertices themselves, to define the triangle. This is necessary in order to avoid the difficulties that would otherwise arise when the theorem names a large number of noncollinear points. If our formal system were a true point geometry, all such constructions would be implicit in the diagram.

are placed on a priority list and developed before any of the others are considered. The second specific heuristic is a routine that assigns a "distance" between each subgoal string and the set of premise strings in some vaguely defined formula space. After those on the priority list have been developed, the next subgoal chosen is that which is "closest" to the premise set in formula space.

It is instructive to examine the machine's behavior in proving complex theorems both with and without the expanded set of semantic heuristics. For the theorem "Two vertices of a triangle are equidistant from the median to the side determined by those vertices," the machine finds a proof in about eight minutes with the basic heuristics alone (see Fig. 6 in Appendix C). The expanded set of heuristics produces a proof in one minute. In addition, the second proof is quite short and to the point, while the first proof meanders blindly about the direct path to the goal before reaching it.

Reflecting the greater efficiency with which the machine attacked the problem in the second trial, only four circuits of the subgoal-generating loop were required compared with twenty-four circuits required without the extended heuristics. Twenty-one intermediate subgoals were generated, compared with sixty-one in the first case, and the problem-solving graph extended to a depth of only three levels, rather than twelve levels for the proof with basic heuristics alone.

For a particular case of a problem taken from a Brooklyn technical high school final examination in plane geometry a solution was found with the extended heuristics in less than five minutes. With the basic heuristics alone, the machine exhausted its working storage in half an hour without having completed the problem. On the other hand, there are problems for which the machine achieves no net gain by applying the additional heuristics. The theorem: "Diagonals of a parallelogram bisect one another" was proved in about three minutes in either mode. The proofs produced in each trial were equivalent, though not the same. A Brooklyn technical high school final examination supplied an example of an intermediate case, where the machine found identical proofs in both modes, but took almost three times as long with the basic heuristics alone (eight minutes, compared with three minutes with extended heuristics). We shall undoubtedly encounter cases for which the application of the extended set will result in a net loss of efficiency, although none has appeared yet in our limited tests.

Conclusion

It is well at this point in our discussion to reemphasize the fact that the object of this research has not been the design of a machine capable of proving theorems in Euclidean plane geometry, or even one able to prove theorems in some undecidable system such as number theory. We are, rather, interested in understanding the use of heuristic methods (or strategies) by machines for the solution of problems that would otherwise be inaccessible to them. Theorem-proving machines in themselves are objects of much interest to mathematicians and logicians, and important work at IBM is being done on this approach by Wang and by Gilmore. Wang (1960*a*) has written a program for the IBM 704 that is able to prove all theorems in propositional logic offered by Russell and Whitehead in the *Principia Mathematica,* whereas the Logic Theorist could master only about 38 of the 52 theorems appearing in chap. 2 of that volume. Also, the time required by the latter machine was far in excess of that used by the former. Newell, Shaw, and Simon, however, were interested in heuristic methods, whereas Wang, and also Gilmore, whose machine deals with the first order predicate calculus, are searching for algorithms, which, though less than a decision procedure, will produce "interesting" proofs within a reasonable amount of time. Both Wang and Gilmore find that for more complex formal systems, heuristics are required (they prefer the word "strategies") to make their algorithms sufficiently selective to produce, within acceptable bounds on space and time, proofs of any great interest.

The work of Wang and Gilmore is most relevant to a new branch of applied logic first characterized by Wang. He names this discipline "inferential analysis," and defines it to include "treatment of proofs as numerical analysis does calculations" (1960*a*). The results of inferential analysis are expected to "lead to mechanical checks of new mathematical results," and ultimately "lead to proofs of difficult new theorems by machine." The present author feels that inferential analysis is relevant, too, to the problem of intelligent behavior in machines. An automaton confronted with the real world, however, will certainly have to rely heavily on heuristics, for the unorthodox formal systems describing its environment will probably be far from amenable to the traditional methods of mathematical logic.

In conclusion, we should like to specify the course of this research for the immediate future. The machine described above is purely a problem-solving system. Except for the annexation of new theorems to the list of axioms, its structure is static. A sequence of practice problems given to the machine will not improve its performance unless a usable theorem is among them. Because it is incapable of developing its own structure, the machine will always be limited in the class of problems it can solve by the initial intent of the designer. It seems that the problem of designing a more general problem-solving machine will be enormously greater than that of designing one not so intelligent but with the capacity to learn.

An immediately obvious approach to the problem of introducing learning into the geometry machine is to allow the machine to adjust all of the parameters that determine its specific semantic heuristics, maximizing the predicted utility of those subgoals that prove to be useful in practice. The machine will thus improve the match of its heuristic filters to the class of problems considered interesting enough to be presented to it for solution. Of greater significance would be the introduction of routines enabling the machine to recognize recurring patterns in its proof-search procedure. Once discovered, such a pattern would enable the machine to construct its own heuristics designed to induce a repetition of the pattern in later proofs. For example, the machine might notice that certain classes of premise strings are regularly followed by the same first step in a proof. The heuristic derived from this pattern would search the premises for such strings and perform the first deduction before starting on the problem-solving graph. The difficult subject of abstract pattern recognition must be understood first, however, and the transformation of pattern to effective heuristic is by no means trivial. But whatever approach to learning is considered most worthwhile to explore, the geometry machine should serve as an excellent framework within which the explorations may be pursued.

Appendix A

Premises

Angle ABD equals angle DBC
Segment AD perpendicular segment AB
Segment DC perpendicular segment BC

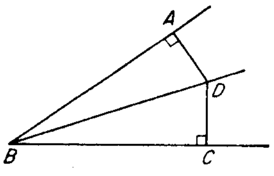


Figure 4.

Definition

Right angle DAB
Right angle DCB

Syntactic Symmetries

CA, BB, AC, DD

Goals

Segment AD equals segment CD

Solution

Angle ABD equals angle DBC
  Premise
Right angle DAB
  Definition of perpendicular
Right angle DCB
  Definition of perpendicular
Angle BAD equals angle BCD
  All right angles are equal

Segment DB
  Assumption based on diagram
Segment BD equals segment BD
  Identity
Triangle BCD
  Assumption based on diagram
Triangle BAD
  Assumption based on diagram
Triangle ADB congruent triangle CDB
  Side-angle-angle
Segment AD equals segment CD
  Corresponding elements of congruent triangles are equal

Total elapsed time = 0.3200 minute

Appendix B

Premises

Quad-lateral ABCD
Segment BC parallel segment AD
Segment BC equals segment AD

CA	BA	DA
DB	AB	CB
AC	DC	BC
BD	CD	AD

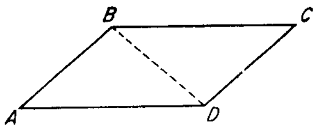


Figure 5.

Goals

Segment AB equals segment CD

I am stuck, elapsed time = 0.88 minute

Construct segment DB
Add premise segment DB
Restart problem

Solution

Segment BC parallel segment AD
  Premise
Opp-side CADB
  Assumption based on diagram
Segment DB
  Premise
Angle ADB equals angle CBD
  Alternate interior angles of parallel lines

Segment BC equals segment AD
  Premise
Segment BD equals segment DB
  Identity
Triangle CDB
  Assumption based on diagram
Triangle ABD
  Assumption based on diagram
Triangle ABD congruent triangle CDB
  Side-angle-side
Segment AB equals segment CD
  Corresponding elements of congruent triangles

Total elapsed time = 4.06 minutes

Appendix C

Premises

Triangle ABC
Precedes BMC
Segment BM equals segment MC
Precedes ADM
Precedes DME
Segment BD perpendicular segment AM
Segment CE perpendicular segment ME

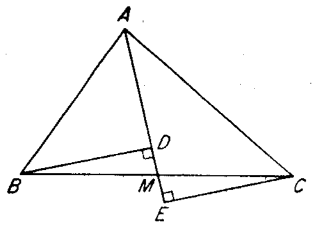


Figure 6.

Definition

Right angle MEC
Right angle BDM
Right angle BDA
Angle BDA equals angle BDM

No syntactic symmetries

Goals

Segment BD equals segment EC

Solution

Precedes BMC
  Premise
Segment EC
  Assumption based on diagram

Angle ECM equals angle BCE
  Same angle
Precedes DME
  Premise
Angle CED equals angle MEC
  Same angle
Right angle BDM
  Definition of perpendicular
Right angle MEC
  Definition of perpendicular
Angle BDM equals angle CEM
  Right angles are equal
Angle CED equals angle BDM
  Angles equal to the same angle are equal
Same side MEDB
  Assumption based on diagram
Same side DACE
  Assumption based on diagram
Collinear EDM
  Ordered collinearity implies collinearity
Precedes ADM
  Premise
Precedes EDA
  Combinatorial properties of ordered collinearity
Collinear EDA
  Ordered collinearity implies collinearity
Angle BDE equals angle AEC
  Different names for equal angles
Precedes EMA
  Combinatorial properties of ordered collinearity
Angle CEM equals angle AEC
  Same angle
Angle BDE equals angle MEC
  Angles equal to the same angle are equal
Angle CED equals angle EDB
  Angles equal to the same angle are equal
Opp side CBED
  Assumption based on diagram
Segment ED
  Assumption based on diagram
Segment EC parallel segment BD
  Segments are parallel if alternate interior angles are equal
Opp side EDCB
  Assumption based on diagram

Segment CB
  Assumption based on diagram
Angle BCE equals angle DBC
  Alternate interior angles of parallel lines
Angle ECM equals angle DBC
  Angles equal to the same angle are equal
Same side CMBD
  Assumption based on diagram
Same side MBEC
  Assumption based on diagram
Collinear CMB
  Ordered collinearity implies collinearity
Angle DBM equals angle BCE
  Different names for equal angles
Angle MBD equals angle MCE
  Angles equal to the same angle are equal
Angle DMB equals angle EMC
  Vertical angles
Segment BM equals segment MC
  Premise
Triangle BDM
  Assumption based on diagram
Triangle CEM
  Assumption based on diagram
Triangle BDM congruent triangle CEM
  Angle-side-angle
Segment BD equals segment EC
  Corresponding elements of congruent triangles

Total elapsed time = 8.08 minutes

WITH BASIC HEURISTICS

Solution

Precedes DME
  Premise
Precedes BMC
  Premise
Angle DMB equals angle EMC
  Vertical angles
Right angle BDM
  Definition of perpendicular
Right angle MEC
  Definition of perpendicular

Angle BDM equals angle CEM
  Right angles are equal
Segment BM equals segment MC
  Premise
Triangle CEM
  Assumption based on diagram
Triangle BDM
  Assumption based on diagram
Triangle BDM congruent triangle CEM
  Side-angle-angle
Segment BD equals segment EC
  Corresponding elements of congruent triangles

Total elapsed time = 1.06 minutes

WITH EXTENDED HEURISTICS

EMPIRICAL EXPLORATIONS OF THE GEOMETRY-THEOREM PROVING MACHINE

by H. Gelernter, J. R. Hansen, & D. W. Loveland

Introduction

In early spring, 1959, an IBM 704 computer, with the assistance of a program comprising some 20,000 individual instructions, proved its first theorem in elementary Euclidean plane geometry (Gelernter, 1959*b*). Since that time, the geometry-theorem proving machine (a particular state configuration of the IBM 704 specified by the afore mentioned machine code) has found solutions to a large number of problems[1] taken from high-school textbooks and final examinations in plane geometry. Some of these problems would be considered quite difficult by the average high-school student. In fact, it is doubtful whether any but the brightest students could have produced a solution for any of the latter group when granted the same amount of prior "training" afforded the geometry machine (*i.e.*, the same vocabulary of geometric concepts and the same stock of previously proved theorems).

The research project which had as its consequence the geometry-theorem proving machine was motivated by the desire to learn ways to use modern high-speed digital computers for the solution of a new and difficult class of problems; a class heretofore considered to be beyond the capabilities of a finite-state automaton. In particular, we wished to make our computer perform tasks which are generally considered to require the intervention of human intelligence and ingenuity for their successful completion. The reasons behind our choice of theorem proving in geometry as a representative task are set forth in detail in an earlier study (1958). We

[1] More than fifty proofs are on file at the present time.

only remark here that problem-solving in geometry satisfies our definition of an intellectual activity, while being at the same time especially well suited to the approach we wished to explore. The fact that geometry is decidable is irrelevant for the purpose of our investigation. The methods employed by the machine are suitable as well for the proof of theorems in systems for which no decision algorithm can exist.

We shall not labor the question as to whether our machine is indeed behaving intelligently in performing a task for which humans are credited with intelligence. The psychologists offer us neither aid nor comfort here; they have yet to satisfactorily characterize such behavior in humans, and have rarely considered the abstract concept of intelligence independent of its agent. In the final analysis, people are occasionally observed to do things that may best be described as intelligent, however vague the connotations of the word. These are, in general, tasks involving highly complex decision processes in a potentially infinite and uncontrollable environment. We should be most happy to have our machine duplicate this kind of behavior, whatever label is affixed to it.

Heuristic Programming and the Geometry Machine

The geometry machine is able to discover proofs for a significant number of interesting theorems within the domain of its *ad hoc* formal system (comprising theorems on parallel lines, congruence, and equality and inequality of segments and angles) without resorting to a decision algorithm or exhaustive enumeration of possible proof sequences. Instead, the theorem-proving program relies upon heuristic methods to restrain it from generating proof sequences that do not have a high *a priori* probability of leading to a proof for the theorem in question.

The general problem of heuristic programming has been discussed by Minsky (1959*a*) and Newell, Shaw, and Simon (1959*a*). The particular approach pursued by the authors has been described at length in the papers to which we have already referred (Gelernter et al., 1958, 1959*b*). We shall therefore defer to the presentation of the machine's detailed results in the full study summarized here for a description of how these results were achieved. It should be recorded here, however, that the geometry machine operates principally in the analytic mode (reasoning backward). At each stage of the search for a proof, a goal exists which must be "connected" with the premises for the problem by a bridge of axioms and previously established theorems of lemmas. If the connection cannot be made directly, then a set of "subgoals" is generated and the process is repeated for one of the subgoals. Heuristic rules are used to reject subgoals that are not likely to prove useful, to select one from those remaining to work on, and to choose particular axioms and theorems to use in generating new subgoals. The machine does depart from this procedure in a number of circumstances (in setting up an indirect proof, for example), but these cases account for only a small fraction of the total search time.

The computer program itself was written within the framework of the so-called Newell-Shaw-Simon list memory (1957*b*). In order to ease the task of writing so massive and complex a machine code, a convenient special-purpose list processing language was designed to be compiled by the already available FORTRAN system for the IBM 704 computer (Gelernter et al., 1960*b*). The authors feel that had they not made free use of an intermediate programming language, it is likely that the geometry program could not have been completed.

Summary of Results

Since its initial solo performance, the geometry machine has existed in several different configurations. In its earliest and most primitive form, the system was equipped with a single major semantic heuristic.[2] That first system was, however, able to prove a large number of interesting, though admittedly simple theorems in elementary plane geometry.[3] The heuristic rule in question, which is independent of the particular formal system under consideration, may be described in the following way. All subgoal formulas that are generated at a given stage of the proof search are interpreted in a model of the formal system; in our case, the model is a diagram, a formal semantic interpretation. If the interpreted subgoal is valid in the diagram, it is accepted as a possible step in the proof, provided that it is noncircular (Gelernter, 1959*a*). Otherwise, it is rejected.

As an experiment, a number of attempts were made to prove extremely simple theorems with the latter heuristic "disconnected" from the system (*i.e.,* all noncircular subgoals generated were accepted). In each case, the computer's entire stock of available storage space was quickly exhausted by the initial several hundreds of first level subgoals generated, and, in fact, the machine never finished generating a complete set of first level subgoals. We estimate conservatively that on the average, a number of the order of 1000 subgoals are generated per stage by the decoupled system. If one compares the latter figure with the average of 5 subgoals per stage accepted when the diagram is consulted by the machine, it is easy to see that the use of a diagram is crucial for our system. (Note that the total number of subgoals appearing on the problem-solving graph grows exponentially with the number accepted per stage.)

Since the procedure described above is a heuristic one, errors are oc-

[2] A semantic heuristic is one based on an interpretation of the formal system rather than on the structure of the strings within that system.

[3] A number of these proofs are reproduced in Gelernter, 1959*b*.

casionally made in the selection or rejection of formulas as subgoals. The diagram is made available to the machine in coordinate representation to finite precision. Formulas are interpreted by transforming them into an appropriate calculation on the numerical coordinates representing the point variables. For example, to check the validity of a statement concerning the equality of two segments, the length of each segment in the figure is calculated, and they are then compared to a certain preassigned number of decimal places. If, instead, the statement concerned parallel segments, the slopes would be calculated and compared. In a small number of cases, round-off error has propagated beyond the allowed value, so that valid subgoals were rejected, or invalid ones accepted. It is important to point out, however, that in no case could this effect result in a false proof. Where valid subgoals were rejected, the machine found alternate paths to the solution. Where invalid ones were accepted, the machine failed, of course, to establish them within the formal system. In the worst possible case, the interpretation error could prevent the computer from finding any solution at all, but never could it lead to an invalid proof.

It should be clear at this point that the diagram is used only to guide the search for a proof by supplying yes or no answers to questions of the form: "Is segment AB equal to segment CD in the figure?", or "Is angle ABC a right angle in the figure?". There is no direct link between the diagram and the formal system in the geometry machine. The behavior of the machine would not be changed if the coordinate representation were replaced by a device capable of drawing figures on paper and scanning them.

In the basic theorem-proving system described above, after a set of subgoals has been generated, each member of the set is explored in order. The next subgoal in line is not examined until the one preceding it has been followed down to a dead end. Too, in generating the next level for a given subgoal, every applicable theorem available is pressed into service.

This system was soon extended by the introduction of selection heuristics for both subgoals and subgoal-generating theorems. The subgoal selection heuristic assigns a "distance" between each subgoal string and the set of premises in a vaguely defined *ad hoc* formula space. At each stage, the next subgoal selected is that which is "closest" to the premises in formula space. The generator selection routine recognizes certain classes of subgoals that are usually established in one step. For such "urgent" subgoals, the appropriate generator is withdrawn immediately, and an attempt is made for a one-step proof (of that particular subgoal) before generating the full set for that formula.

The extended system is able to prove a number of somewhat more difficult theorems that are beyond the capacity of the basic machine. For those problems within the range of both systems, the former is, on the

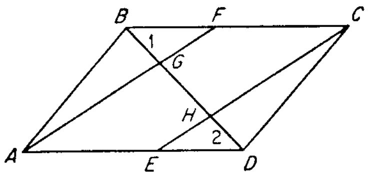


Figure 1.

average, about three times faster, and generates about two-thirds the total number of subgoals in half as many subgoal generation cycles as required by the basic system. The average depth of the problem-solving graph for the refined system, about seven to nine levels, is two-thirds the average depth for the basic system.

By the addition of a simple construction routine, the theorem-proving power of the machine is expanded to include an entirely new class of problem, hitherto logically unattainable. The routine, called upon only when all other attempts have failed, allows the machine to join two previously unconnected points in the diagram, and extends the newly created segment to its intersections with all other segments in the figure. The new segment, when it intersects previously given ones, introduces new points into the problem which are named by the machine and become part of the problem system.

At this stage in its development, the geometry machine was capable of producing proofs that were quite impressive (Appendix 1).[4] Its performance, however, fell off rapidly as the number of points in the diagram increased. This effect was due largely to the fact that unlike humans, who generally identify angles visually by their vertices and rays, the computer specifies an angle by a predicate on three variables, the vertex and a point on each ray. Consequently, the equality of angles 1 and 2 in Fig. 1 may be represented in thirty-six different ways, since each angle has six different names. Formal rigor demands, too, that the equality of angles ADH and EDG, for example, be proved rather than taken for granted. It should be clear that where the condition above exists, the search for a proof quickly bogs down in a mass of uninteresting detail.

In the current system, the angle problem is solved by allowing the machine to use the diagram to identify a given angle with its full set of names, and to assume the equality relationship between different names for the same angle, as does its human counterpart. The geometry machine in its present configuration is able to find proofs for theorems of the order of difficulty represented by the following:

Theorem: If the segment joining the midpoints of the diagonals of a

[4] In the proofs appended to this paper, the nonobvious predicates have the following interpretations:

OPP-SIDE XYUV	Points X and Y are on opposite sides of the line through points U and V.
SAME-SIDE XYUV	Points X and Y are on the same side of the line through points U and V.
PRECEDES XYZ	Points X, Y, and Z are collinear in that order.
COLLINEAR XYZ	Points X, Y, and Z are collinear.

trapezoid is extended to intersect a side of the trapezoid, it bisects that side (Appendix 2).

Limitations of the System

It will be immediately evident to those familiar with the properties of formal logistic systems that unless a construction which generates a new point is introduced by the machine, all problems are solved within the framework of a propositional calculus, however complex its structure. Although the machine's present construction routine can and does generate new points, we could not expect our results to be of great interest to logicians until a full set of possible constructions (corresponding to a complete set of existentially quantified axioms) is made available to the system to abet its search for a proof.

An equally serious limitation on the formal generality of the theorem-proving machine is imposed by our method for determining the well-formedness of strings within the logical system. In order to attain the necessary speed and efficiency in processing, well-formed formulas are defined by schema rather than recursively. The kind of statement that can be made in the system is then determined by the schema available to the machine. The practical effect of this loss in generality is to restrict rather severely the freedom with which algebraic statements in geometry may be manipulated.

In addition to the above, there are a number of nonessential bounds on the theorem-proving ability of the machine. These are a consequence of the limited speed and memory capacity of the computer for problems of such highly combinatorial character. Improvements in either of the above will be immediately effective in extending the class of machine-solvable problems in both quantity and difficulty.

Conclusion

The initial goal of our research program in machine intelligence has been attained. If the interrogator were to restrict his probing to the area of theorem-proving in elementary Euclidean plane geometry, our machine could be expected to give an excellent account of itself in competition with a human in Turing's well-known "imitation game" (1950). Of course there are many other problem areas (solving arithmetic problems, for example) where computers have always been able to compete successfully with humans. The significant point is that a knowledgeable interrogator would certainly avoid such areas in his questioning, while he might well (until now, at any rate) introduce a plane geometry problem in a calculated attempt to separate the men from the machines.[5] Although the stage is now set for the argument that any distinct area of human intellectual activity will in the same way succumb to the inexorable logic of electrons, switches, and gates, we defer to our philosopher colleagues for debate on the implications of that contention, at least until the time that computers have been programmed to consider such issues.

There are a number of consequences of our work that are, fortunately, more concrete than that alluded to above. Perhaps the most important are those relating to inferential analysis, a new branch of applied logic first characterized by Wang (1960*a*). Inferential analysis "treats proofs as numerical analysis does calculations," and is expected to "lead to mechanical checks of new mathematical results" and, more important, "lead to proofs of difficult new theorems by machine." It is expected that our techniques for the manipulation and efficient search of problem-solving trees and our results concerning syntactic symmetry will prove to be useful tools in pursuing the goals of inferential analysis.

Contributions have been made, too, in the area of techniques for computer implementation of complex information processes. Results pertaining to the design and use of intermediate languages for the specification of list manipulation processes have been reported elsewhere (Gelernter et al., 1960*b*). The latter work indicates clearly the requirements of a digital computer system designed for optimum execution of such list processes. In brief, a list processing computer should possess hardware facilities for:

1. Generalized indirect addressing; specified in the indirectly addressed instruction to arbitrary depth and in arbitrary order from either the left or the right field of a two-address register,
2. Effective address recovery; making available the terminal content of the address register (the final address in a long and complex indirect address chain, for example) as the address field for a subsequent operation,
3. Field logic; a greatly expanded set of interfield operations within a full register sectioned according to some previously established convention, and
4. List search operations; a list equivalent of the conventional table look-up instruction.

The bulk storage input-output requirements for a list processing computer are severe, and are not included in the enumeration above. The system

[5] It may be argued (and undoubtedly, it *will* be argued) that the truly knowledgeable interrogator, cognizant of the decidability of geometry, would certainly avoid this area as well, perhaps preferring the manifestly undecidable parts of the predicate calculus or number theory to effect the distinction between man and machine. We recall here that our methods are independent of the decidability of the formal system, and, in fact, Wang (1960*a*) and Gilmore (1960) have developed proofs for theorems in the undecidable area of the predicate calculus.

design of a digital computer for the manipulation of list structures will be described in detail in a subsequent paper.

Finally, we consider the implications of our work for the basic problem of machine intelligence. The geometry machine, we feel, offers convincing evidence of the power and fruitfulness of heuristic programming for the solution of problems of a certain class by computer. In our experience, the theorem-proving power of the machine has often been extended by the addition of a single heuristic to a degree equivalent to a three-to-fivefold increase in the speed or storage capacity of the computer.

Our program has proved to be disappointing as a tool for the study of the more elementary trial-and-error types of machine learning, largely because of the rather low rate at which it accumulates experience. It is reasonable to expect, however, that the geometry machine might yet be pressed into service in an investigation of the higher, conceptual types of machine learning, providing that one will someday know how to formulate the problem.

If nothing else, our work offers some qualitative indication of the order of magnitude of difficulty for problems that could be expected to yield to contemporary computer technology. Three years ago, the dominant opinion was that the geometry machine would not exist today. And today, hardly an expert will contest the assertion that machines will be proving interesting theorems in number theory three years hence.

Appendix 1

Premises

Quad-lateral ABCD
Point E midpoint segment AB
Point F midpoint segment AC
Point G midpoint segment CD
Point H midpoint segment BD

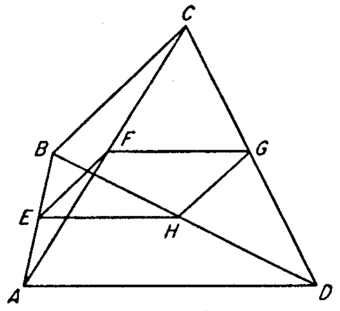


Figure 2.

To Prove

Parallelogram EFGH

Syntactic Symmetries

BA, AB, DC, CD, EE, HF, GG, FH, CA, DB, AC, BD, GE, FF, EG, HH, DA, CB, BC, AD, GE, HF, EG, FH

Proof

Segment DG equals segment GC
  Definition of midpoint

Segment CF equals segment FA
  Definition of midpoint
Triangle DCA
  Assumption based on diagram
Precedes DGC
  Definition of midpoint
Precedes CFA
  Definition of midpoint
Segment GF parallel segment AD
  Segment joining midpoints of sides of triangle is parallel to base
Segment HE parallel segment AD
  Syntactic conjugate
Segment GF parallel segment EH
  Segments parallel to the same segment are parallel
Segment HG parallel segment FE
  Syntactic conjugate
Quad-lateral HGFE
  Assumption based on diagram
Parallelogram EFGH
  Quadrilateral with opposite sides parallel is a parallelogram

Total elapsed time = 1.03 minutes

Appendix 2

Premises

Quad-lateral ABCD
Segment BC parallel segment AD
Point E midpoint segment AC
Point F midpoint segment BD
Precedes MEF
Precedes AMB

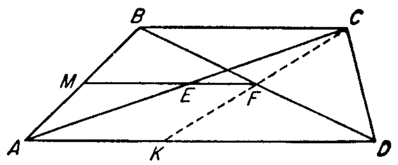


Figure 3.

To prove

Segment MB equals segment MA

No Syntactic Symmetries

I Am Stuck, Elapsed Time = 8.12 Minutes
Construct segment CF
Extend segment CF to intersect segment AD in point K

Add to Premises the Following Statements

Precedes CFK
Collinear AKD

Proof

Segment BC parallel segment AD
  Premise
Collinear AKD
  Premise
Segment KD parallel segment BC
  Segments collinear with parallel segments are parallel
Opp-side KCDB
  Assumption based on diagram
Segment DB
  Assumption based on diagram
Angle KDB equals angle CBD
  Alternate interior angles of parallel lines are equal
Precedes CFK
  Premise
Precedes DFB
  Definition of midpoint
Angle KFD equals angle CFB
  Vertical angles are equal
Segment DF equals segment FB
  Definition of midpoint
Triangle FDK
  Assumption based on diagram
Triangle FBC
  Assumption based on diagram
Triangle FDK congruent triangle FBC
  Two triangles are congruent if angle-side-angle equals angle-side-angle
Segment KF equals segment CF
  Corresponding segments of congruent triangles are equal
Segment CE equals segment EA
  Definition of midpoint
Triangle AKC
  Assumption based on diagram
Precedes CEA
  Definition of midpoint
Segment EF parallel segment AK
  Segment joining midpoints of sides of triangle is parallel to base
Segment EF parallel segment KD
  Segments collinear with parallel segments are parallel
Segment FE parallel segment BC
  Segments parallel to the same segment are parallel

Precedes MEF
  Premise
Collinear MEF
  Ordered collinear points are collinear
Segment FM parallel segment BC
  Segments collinear with parallel segments are parallel
Segment FM parallel segment DA
  Segments parallel to the same segment are parallel
Triangle DBA
  Assumption based on diagram
Precedes AMB
  Premise
Segment MB equals segment MA
  Line parallel to base of triangle bisecting one side bisects other side

Total elapsed time = 30.68 minutes

section 4

Two Important Applications

What good is artificial intelligence research? Can the devices and techniques developed be used in the solution of "real" problems? These are questions which a practical man, an engineer, say, or an operations research analyst, might reasonably ask.

The reports in this section describe heuristic programs which solve complex problems in important areas of application. One program handles integration problems in the elementary calculus. The other handles problems of assembly line balancing in a manufacturing process. The problems have these features in common:

1. Both are problems which are moderately difficult for intelligent human beings with college training.
2. Both are readily attackable by heuristic methods, and these methods closely resemble the methods used by intelligent human beings to solve the problems.
3. It is economically feasible to use programs to solve these problems. In most cases, it would be difficult to hire a man with comparable skill to solve the problems as cheaply. The time required by an IBM 7090 to solve the problems is generally much shorter than would be required by the average human problem-solver skilled in the problem area. This is true in spite of the fact that present-day computers were not designed with heuristic programming applications in mind and hence existing computer languages well suited to heuristic programming use digital computers quite inefficiently.

The assembly line balancing program developed by F. Tonge is an application of heuristic programming to an important management science problem. Balancing an assembly line involves finding an efficient arrangement of workers, tasks, and work stations so as to maximize the rate of assembly or minimize the number of workers needed for a given rate of assembly. More or less "straightforward" procedures for doing this have been devised, but they are generally not practical, since they involve the enumeration of a very large number of "tries"—combinations of the work elements. Tonge's program differs from these in that it employs a variety of line balancing heuristics—"tricks of the trade"—to simplify the constraints, to structure the assignment part of the problem, and to carry out the actual computations.

A significant feature of Tonge's program is its "level-of-aspiration" effort-limiting heuristic. In the line balancing problem, the maximum rate of assembly or the minimum number of workers for a given rate can be computed easily, even though the actual assignment cannot. Tonge's program *does not seek the optimal solution but merely one which is "satisfactory," i.e.,* within some (given) percentage of the calculated optimum. The power of this heuristic derives from the facts that, the closer to optimum one requires the solution to be, the more computing effort is needed to find the solution (as one would expect), and that the relationship is strongly nonlinear.

Slagle's SAINT program handles problems in an area familiar to many of us. By now, the elementary calculus is almost a common language among university graduates. The integration of elementary functions (when the answer is not to be found by rote recall or looking in a table of integrals) is not a trivial intellectual task. Many a college sophomore has stayed up half the night searching for the "key" which would unlock the solution to some complicated integral. The successful human performer is generally considered to have not only a wide repertory of "tricks and transformations" he can apply but also a keen "intuitive feel" concerning which tricks to choose and what sequence to apply them in. Finally, analytic integration of elementary functions constitutes a significant portion of the routine mathematics of modern engineering and natural science.

SAINT is a program for performing analytic integration of elementary functions. It uses the same kinds of "tricks" that are taught to and used by students in elementary calculus courses. In its conception and structure, it is a linear descendant of the Logic Theory machine. As with LT, the SAINT program shows that the behavior vaguely labeled "cleverness" or "keen insight" in human problem-solving is really just the result of the judicious application of certain heuristics for narrowing and guiding the search for solutions.

Fred Tonge is on the faculty of the Graduate School of Industrial Administration, Carnegie Institute of Technology.

James Slagle is a member of the staff of the Lawrence Radiation Laboratory, Livermore, California.

SUMMARY OF A HEURISTIC LINE BALANCING PROCEDURE

by Fred M. Tonge

1. Introduction

This study describes a heuristic program for assembly line balancing. We employ heuristic methods because the assembly line balancing problem, like many combinatorial problems, has not been solved in a practical sense by advanced mathematical techniques.

Because this approach does not guarantee an optimum solution, the ultimate measure of a heuristic program is whether it provides better solutions more quickly and/or less expensively than other methods. However, at this early stage in the development of heuristic procedures there is still much to be learned about both the specification and the mechanization on a computer of such procedures. The research reported here was undertaken to explore these questions. No special emphasis was placed on producing an economically competitive program; but the results are sufficiently interesting to report.

Here we summarize the heuristic program developed for assembly line balancing and the operating results obtained with that program. A detailed description of the procedure and further comments on this approach to utilizing digital computers are presented elsewhere (Tonge, 1961*a*).

2. Assembly Line Balancing

In many industries (home appliances, automobiles) the product is assembled on a continuous conveyor line. The elemental tasks making up the assembly operation must be assigned to work stations along the line. (For

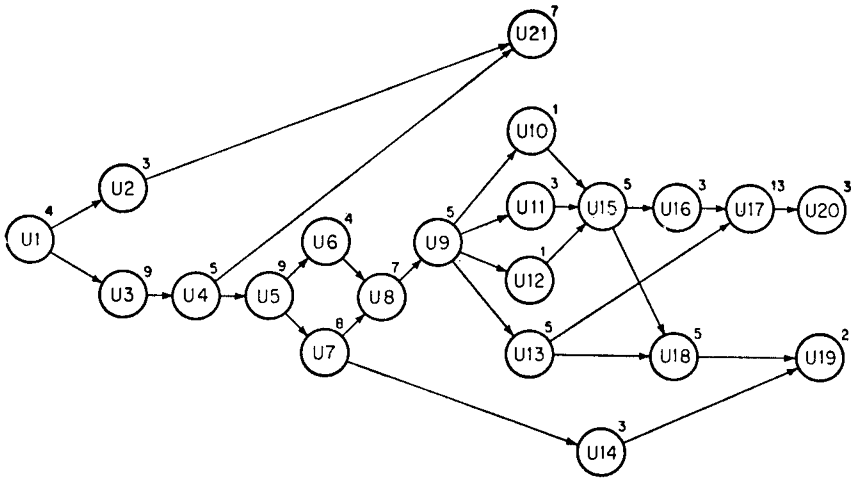


Figure 1. Twenty-one-element problem-directed graph representation.

our purposes, "work station" and "operator" are equivalent.) In the simplest case, each elemental task (also called "task" or "element") is characterized by an operation time per unit of product and a partial ordering relationship with other elemental tasks.

Figure 1 represents one such assembly (taken from Mitchell, 1957). Here, for example, elemental task U5 requires nine time units. It cannot be started until task U4 is completed, and must be completed before either U6 or U7 can be started. The constraint that task U5 must precede task U14 need not be represented explicitly; this information is implied by the sequence U5 → U7 → U14.

In industrial practice the ordering constraints are not explicitly stated. The industrial engineer works directly from bill of material and standard time data for the product and from his own knowledge of manufacturing technology.

A production rate set by management determines the maximum time (cycle time) to be assigned to any work station. That is, hours per shift divided by units per shift determine the maximum time an operator can spend on each unit.

The assembly line balancing problem[1] can be stated as:

Given a production rate (or, equivalently, a cycle time), what is the minimum number of work stations (operators) consistent with the time and ordering constraints of the product?

[1] There are, of course, many variants of the balancing problem, introducing such factors as uncertainty of time values, varying operator capabilities, etc. None of these other constraints are treated here.

TABLE 1

Work station	Total time	Elemental tasks
1	18	U1, U3, U4
2	19	U2, U5, U21
3	15	U6, U7, U14
4	17	U8, U9, U10, U11, U12
5	20	U13, U15, U16, U18, U19
6	16	U17, U20

A closely related problem is:

Given some number of men (work stations), what is the maximum production rate consistent with the time and ordering constraints of the product?

We shall consider the first formulation.

More explicitly, then, the assembly line balancing problem concerns a set of elemental tasks such that:

1. each elemental task requires a known operation time per unit of product, independent of when performed;
2. a partial ordering exists among the elemental tasks.

An optimal solution of the problem consists of an assignment of elemental tasks to work stations such that:

1. each elemental task is assigned to one and only one work station;
2. the sum of the times of all elemental tasks assigned to any one station does not exceed some maximum (the cycle time);
3. the stations thus formed can be ordered such that the partial orderings among elemental tasks are not violated;
4. the number of work stations thus formed is minimized.

A minimal work-station solution to the problem of Fig. 1 (given a cycle time of 20) is shown in Table 1.

Actual instances of the assembly line balancing problem are much more complex than the example given here. A representative problem from the appliance industry would contain nearly a hundred elemental tasks. The industrial engineer balancing a line works from a sheet listing the elements and their operation times and from his knowledge of the manufacturing process. In the absence of any formal procedures, he must rely on judicious use of trial-and-error methods to find an acceptable grouping.

3. *Exhaustive Procedures for Assembly Line Balancing*

Assembly line balancing techniques have received little attention in standard production management literature; most works merely acknowledge that the problem is a common one. However, several management scientists have studied the problem.

Salveson (1955) reviews the industrial setting of the problem and comments on several difficulties associated with line balancing, *e.g.*, determining before solution the minimum number of operators, taking into account variances in operation times. He suggests application of limited combinatorial analysis, but the procedure rests upon previous enumeration of all possible work stations, an enormous task. However, many of Salveson's observations on the nature of the assembly line balancing problem are valid, and this article first brought the problem to the attention of the management science audience.

Jackson (1956) presents an algorithm for systematically enumerating and evaluating possible solutions. Comparisons of this exhaustive method with the heuristic procedure developed here are presented in a later section.

In commenting on Jackson's article, Helgeson and Kwo (1956) suggest an additional criterion—minimizing the variation in work load among stations—in evaluating possible solutions.

Bryton (1954) develops a "convergence" procedure for shifting elements among given work stations so as to increase the maximum attainable production rate. While not directly applicable to the problem stated here, this procedure is useful for equalizing the work load among stations once a solution has been found.

Mitchell (1957) extends Jackson's algorithm to include another common constraint, zoning. By zoning is meant the division of the set of elemental tasks into (possibly overlapping) subsets corresponding to physical constraints on the assembly operation. Zoning of an assembly line may be determined by the position of the product on the conveyor, the layout of the production facility, or both. For example, certain elements may be performed only from the back of the product, or only while it is lying on its side; likewise, some elements may be carried out on a smaller subline joining the main conveyor. The constraint that all elemental tasks assigned to a work station must be in the same zone is added to the definition of a solution.

The methods mentioned above are not practical for large assembly line balancing problems because of their computational requirements. Indeed, no satisfactory general scheme for resolving large combinatorial problems involving partial ordering relations has yet been devised.[2]

4. *Heuristic Programs*

Webster's New International Dictionary of the English Language, 1959, defines the adjective "heuristic" as "serving to discover or reveal." Thus,

[2] However, much unreported research is being done by industrial firms into assembly line balancing and related problems.

by heuristics we mean (after Newell, Shaw, and Simon, 1958*d*) principles or devices that contribute, on the average, to reduction of search in problem-solving activity. The admonitions "Draw a diagram" in geometry, "Reduce everything to sines and cosines" in proving trigonometric identities, or "Always take a check—it may be mate" in chess are all familiar heuristics.

Heuristic problem-solving procedures are procedures organized around such effort-saving devices. A heuristic program is the mechanization on a digital computer of some heuristic procedure. The computer attempts to solve the problem by carrying out the heuristic program. At present this use of digital computers is the only means we have of making explicit the behavior of a complex heuristic procedure in dealing with a large class of problems. The Logic Theorist (Newell, Shaw, and Simon, 1957*a*), the Chess Machine (Newell, Shaw, and Simon, 1958*b*), and the Geometry Machine (Gelernter and Rochester, 1958) are examples of working heuristic programs in other areas.

The distinction between heuristic and non-heuristic problem-solving procedures is often vague. Rather than attempt to specify a rule by which all procedures can be so categorized, we shall cite some common characteristics of existing heuristic procedures:

1. Factorization of the problem into a number of "smaller" problems and subproblems (often through means-end analysis), with a corresponding goal-subgoal organization of behavior. For example, the Chess Machine might realize that it cannot play P-K4 because it would lose an exchange on that square, and consequently sets up the subgoal of first bringing another man to bear on its K4.
2. Use of cues in the environment to determine the particular behavior evoked from a wide set of possible alternatives available to the program. That is, a high degree of interdependence between the specific problem (from a more general class) being considered and the particular problem-solving methods used. Thus, the methods used by the assembly line balancing program for choosing elements to shift between groupings depend on the particular characteristics of those groupings.
3. Use of recursive procedures to bring to bear on subproblems the same repertoire of problem-solving techniques used on the original problem. Thus, the Logic Theorist can use the same "bag of tricks" to prove a derived expression as to prove the initial statement from which the derived expression was produced.
4. No guarantee of a satisfactory solution or, often, of any solution. For example, the Chess Machine, because of time and space limitations, may not be able to consider some promising continuations, including the a postiori optimum one.

Because a heuristic procedure substitutes the effort reduction of its shortcuts for the guaranteed optimal solution of an exhaustive method, the justification of such a program as a problem-solver must be in terms of the number of cases successfully solved and the relative amount of effort involved. In a later section on operating results, the assembly balancing program presented here is compared with several exhaustive procedures.

The set of heuristics outlined here for balancing an assembly line evolved from several sources: discussions with industrial engineers of how they actually balance lines; study of the various papers cited above; explorations with several co-workers, particularly A. Newell, of possible techniques; and finally extensive experimentation with particular instances of the problem.

5. *The Assembly Line Balancing Program*

This heuristic approach to assembly line balancing is based on simplification—sufficient simplification of a complex combinatorial problem that it becomes solvable (in most cases) by simple, straightforward methods.

Two recursively defined routines form the essence of this procedure.[3] Phase I constructs a hierarchy of increasingly simpler line balancing problems by aggregating groups of elements into a single compound element. Each of these compound elements is itself a member of this same class of line balancing problems, since it is made up of elements requiring a given operation time and among whom partial ordering relationships exist.

Phase II solves a simple (small number of elements) line balancing problem by assigning groups of available workmen to elements and then taking as subproblems those compound elements (simple problems in themselves) which have been assigned more than one man.

This approach requires heuristics for aggregating groups of elements into compound elements, for solving the simplified problems thus created, and for reintroducing the detail of the original problem when the simplified version does not yield a solution.

A third phase of the problem-solving process, utilizing virtually the same heuristics as already required, involves "smoothing" the final work load (assigned time) among work stations. Since the greatest total time assigned any work station limits the speed of the line, a smooth balance answers

[3] Several other strategies could be suggested as bases for heuristic approaches to the problem: (1) division of the problem into relatively independent subproblems, solution of these by optimizing techniques, and combination of the subsolutions by heuristic methods; (2) solution of a problem abstracted from the original by replacing all times with one of a single large or a single small value, and then adjustment of the result to fit actual times; (3) development of stations in an order determined by the density of partial ordering constraints, first building in those regions of the problem where there is least freedom among elements. None of these alternative approaches are considered here in any further detail.

the problem "What is the highest production rate achievable with a given number of men?" That is, since both men and time are measured in discrete units, a *nonsmoothed* optimum solution of the problem "given a production rate, minimize the number of men required" need not be an optimum solution of the dual problem "given a number of men, maximize the production rate."

Thus, the general problem-solving scheme calls for:

Phase I. Repeated application of aggregative procedures, creating a hierarchy of simplified line balancing problems ranging in complexity from the initial problem to one containing a single compound element.

Phase II. Recursive application to these simplified problems of a procedure for assigning men to tasks, down to the level of problems whose component tasks require one man each. When the compound elements making up a problem require more men than are available, these elements are broken up and their components regrouped to require fewer men.

Phase III. Smoothing the resulting balance by transferring tasks among work stations until the distribution of assigned time is as even as possible.

The following sections discuss the heuristics entailed in each of these procedures.

Since this approach does not guarantee an optimum solution to the overall line balancing problem, we must have some notion of a satisfactory solution and accept or reject proposed solutions based on this notion. Also, since the solution of a problem does not guarantee that its subproblems will be solved, whatever process generates solutions must remain active until all subproblems have been solved, ready to generate another solution if necessary. The methods by which these requirements are met are indicated below.

6. *Constructing the Hierarchy of Problems*

The ordering constraints present in the problem suggest two natural units of aggregation of elements. Either a completely ordered relationship exists between several elements—as U3 must always precede U4 (Fig. 1)—or no ordering is specified—as U10, U11, U12. We adopt the "chain" and the "set" as basic aggregative units in constructing a simplified problem:

I. A group of adjacent elements whose relative order is completely determined, each except the first having a single direct predecessor and each except the last having a single direct follower, can be replaced by a single compound element, called a *chain*.

II. A group of elements whose relative order is completely unspecified, all having the same direct predecessors and followers, can be replaced by a single compound element, called a *set*.

Thus, U10, U11, U12 in Fig. 1 can be replaced by a set V2, and then V2, U15 can be replaced by a chain V3. These aggregations can be indicated on the original problem, as in Fig. 2*a*. However, it is convenient to represent the aggregations as a branching tree, with each compound element having beneath it the elements of which it is composed.

The tree of compound element V3 is shown in Fig. 2*b*. Note that the time requirement of a compound element is the sum of the times of its components.

We use the term "chain relationship" to indicate that some ordering (possibly indirect) exists between two elements and "set relationship" to indicate that none exists.

While the solution strategy calls for repeated application of the aggregative operations to yield a hierarchy of simplified problems, these two types of aggregation are not sufficient to completely reduce (to a single compound element) most actual problems. We can proceed by first defining more complex aggregative operations and then, if necessary, removing "troublesome" ordering constraints. Proposed solutions must then be checked to see that these missing constraints are not violated.

The one additional compound element introduced to date is the Z.

III. A *Z* is a group of four elements with the two front elements having common predecessors and the other two back elements having common followers. The single direct follower of one front element is one of the back elements; the two direct followers of the other front element are the back elements. The back elements have no other direct predecessors.

An example of a Z and its representation in tree form are given in Figs. 3*a* and 3*b*. (Note that there is a canonical order of subelements in the tree representation of the chain and the Z.)

The recursive procedure carrying out this aggregating process may be applied to any assemblage of elements. By an assemblage of elements we

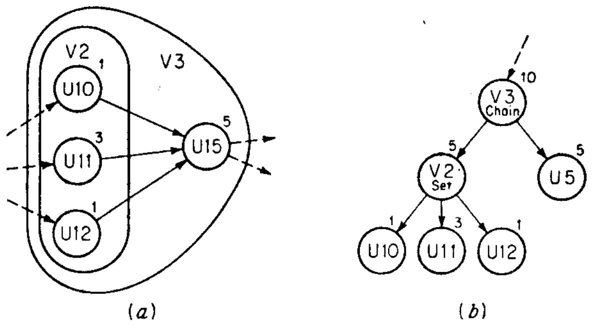


Figure 2. Chain and set aggregations.

here mean those given elements without predecessors within the assemblage (called the front elements) and their direct followers, and their followers' direct followers, and so forth, to and including those elements with no direct followers. Thus, the illustrative problem of Fig. 1 is an assemblage with front element U1.

While there is not space here to present the full details of this procedure, a brief summary follows.

Given an assemblage of elements with a single front element, the routine attempts to create a chain. When an element having several direct followers is encountered, the program first sets up the subproblem of aggregating the assemblage with those several front elements and then applies itself to that problem. (The routine is recursive, so that attempting to solve that subproblem may involve setting up and attempting to solve sub-subproblems and so forth.) If the subproblem is solved successfully, the higher-level problem of creating a chain is continued. If the subproblem fails, or if an element is encountered with direct predecessors outside the assemblage (other than those of the front element), the higher-level problem fails.

Given an assemblage of elements with several front elements, the routine attempts to create a set or, failing that, a Z. If this can be done, the problem becomes one of reducing an assemblage with a single front element. If not, the routine first sets up the independent subproblems of aggregating separate assemblages starting with each of the front elements, and applies itself to each of these subproblems in turn. When these subproblems are concluded, the higher level problem of creating a set or Z is resumed.

Trying to resolve these subproblems independently often reveals complex ordering constraints between them. A Z is then postulated incorporating such constraints. Further constraints that prevent completion of an already postulated Z are relaxed so that the Z can be completed, and the higher level problem is then continued.

Figure 4 is the tree of compound elements constructed by application of this recursive procedure to the problem of Fig. 1. The running commentary of this routine is given in Appendix A; this is a dynamic statement of how

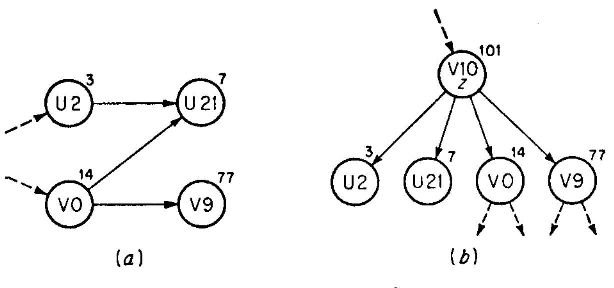


Figure 3. Z aggregations.

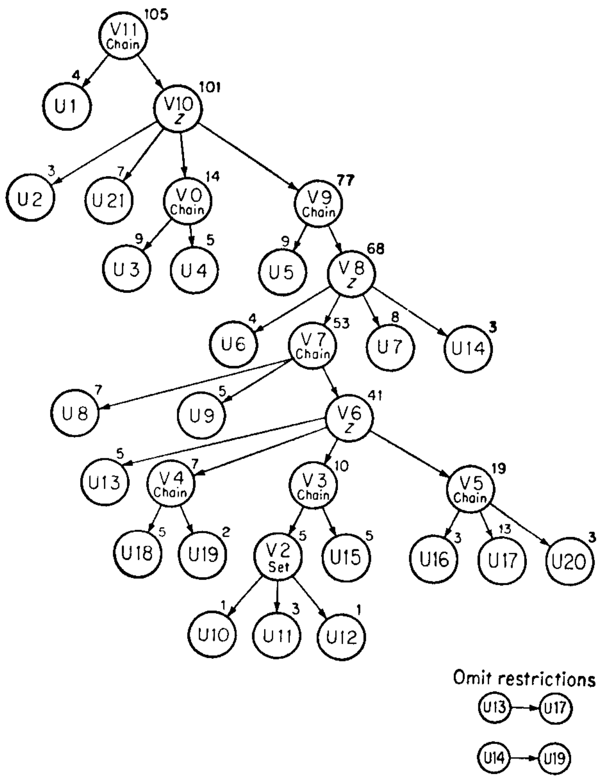


Figure 4. Twenty-one-element problem—Phase I output.

the routine is proceeding. It is instructive to follow through this "protocol" while viewing Figs. 1 and 4. This first phase of the problem-solving process need be carried out only once for a given product (set of constraints). The resulting hierarchy is supplied as an input to the second phase of the process.

7. *Grouping Tasks into Work Stations*

Inputs to the second phase, grouping tasks into work stations, are: the problem hierarchy as developed in Phase I; a cycle time determined by the required production rate; and a "per cent usable time" supplied as a guide for setting up and accepting potential work stations. Per cent usable time is an estimate (at present made by the user) of how close to the required station the task elements can be grouped. This measure reflects the structure and time values of the particular problem being solved.

The assignment routine consists of a simple recursive procedure for allocating men (work stations) to groups of tasks and a set of procedures for regrouping the tasks when necessary. These regrouping heuristics are described in the following section.

The recursive routine for allocating men to groups of tasks proceeds as follows:

1. An initial estimate is made of the number of men required to meet the given production rate, given a per cent usable time (per cent effectiveness). This number of men is assigned to the "top" compound element, representing a single grouping of all task elements. The recursive routine described in step 2 is then applied to that element. If the routine succeeds, the line has been balanced and Phase II ends. If the routine fails, the number of men assigned to the top element is increased and the procedure repeated.

2. (*a*) If the element being considered has been assigned more than one man, these available men are allocated among the direct components of the element according to their time requirements. This recursive routine is then applied to each of these component elements in turn. If the routine successfully solves each of the subproblems, then the problem of grouping tasks into work stations is also solved at this level and the routine terminates. If the available men cannot be allocated among the direct components of the element in question—if, considered independently, they require more men than are available—then regrouping procedures are called upon to shift tasks among these groupings so that they are independently solvable. If such a regrouping cannot be found, control is returned to the next higher level, signaling failure to solve the subproblem.

If the recursive routine should fail to handle one of the components below this level and report back failure, excess elements from the failing subproblem are shifted to another grouping or, failing that, the regrouping procedures are activated to produce another set of independent groupings. Again, if solvable subproblems cannot be found, control is returned to the next higher level with a failure signal.

(*b*) If the element being considered has been assigned a single man, additional elements are added to it if necessary to bring near the maximum allowable size and the grouping is marked as a work station. Control then returns to the next higher level, signaling success.

Thus, this recursive routine, like that of Phase I, deals with examples of a particular class of problems. It solves a problem by setting up within it (and solving in the same way) other "easier" problems of the same class whose solutions can be combined to solve the larger problem also. In particular, Phase I of the line balancing program makes use of natural groupings of the elemental tasks to build up a hierarchy of simplified problems, and Phase II attempts to balance the line using these groupings as complete units, transferring tasks among these groupings only when the simple allocation scheme fails.

8. *Regrouping Procedures*

Five regrouping procedures—direct transfer, trading, sequential grouping, complete grouping, and exhaustive grouping—are available to the Phase II recursive routine when its simple scheme for allocating available men fails. Which of these regrouping heuristics will be called upon, and in what order, is determined by characteristics of the compound elements being regrouped. These procedures are also used by Phase III of the assembly line balancing program in smoothing a proposed line balance.

All five methods make use of a single recursive routine that scans a given part of the hierarchy of elements (the "problem tree") and generates a sequence of groups of elements lying within certain specifications (maximum total time of group, minimum total time of group, from front or from back). This routine proceeds by building a first group, always taking the largest element acceptable, and then constructs further groups by eliminating from consideration (or breaking down into components, if a compound element) successive elements of that group and applying itself to the remaining elements of the given part of the hierarchy. Thus, if requested to produce groups totaling at least 8 but not more than 16 time units from the back of compound element V6 (see Fig. 4), the routine would generate the sequence (V4, U13, U20), and (U20, U17). The routine generates each element of this sequence as requested, remembering where it is in the sequence to be able to generate the next one.

The first three methods—direct transfer, trading, and sequential grouping—are used to set up independently solvable subproblems which can then be handled by the Phase II recursive routine.

Direct transfer is applied when only two components are involved. This method tries simply to transfer elements from one component to the other and thus reduce the number of men required by a straightforward totaling of whole men.

Thus, if V11 in Fig. 4 had been assigned six men (given a cycle time of 20), the direct transfer procedure would first attempt to shift elements totaling at least 1 but not more than 16 time units from V10 to U1, and would in fact shift V0 (14 time units).

Trading also is applied only to two components, and assumes that direct transfer has been attempted without success. Trading tries to regroup by shifting an element larger than the acceptable limit from one component in exchange for smaller elements (in a set relationship with the first element shifted) from the other component.

An example of this method is cited in the next section.

Sequential grouping is used when there are several components, and attempts to construct an acceptable work station from the front of the given group of components. If the remaining components can be handled by the remaining men available, the method is considered successful. If not, an attempt is made to construct another work station from the back of the component group, and a similar test is made.

Suppose, for example, that V7 in Fig. 4 had been assigned three men (cycle time 20). Since V7 is made up of three components requiring one, one, and three men, regrouping procedures would be evoked. Sequential grouping would first group together U8, U9, and V2 and then, since the remaining component V6 would now total 36 time units, requiring only the two remaining available men, the method would be successful.

The remaining two regrouping procedures, complete grouping and exhaustive grouping, try to completely solve the subproblems remaining. They may be regarded as "last-ditch" methods at any particular level.

Complete grouping attempts, first from the front of the component group and then from the back, to construct work stations until all task elements are grouped. If at any time the method cannot construct a station such that the remaining elements total less than can be handled by the remaining men, the method fails.

Exhaustive grouping generates all possible (independent) first work stations, then all possible following work stations for each of these. This method is, in fact, the exhaustive algorithm suggested by Jackson (1956). Because of the comparatively large amount of effort required to do an exhaustive grouping, this procedure is used only when two men are to be assigned. The method does furnish to the next higher level Phase II routine (on failure) the best groupings it was able to construct from the front

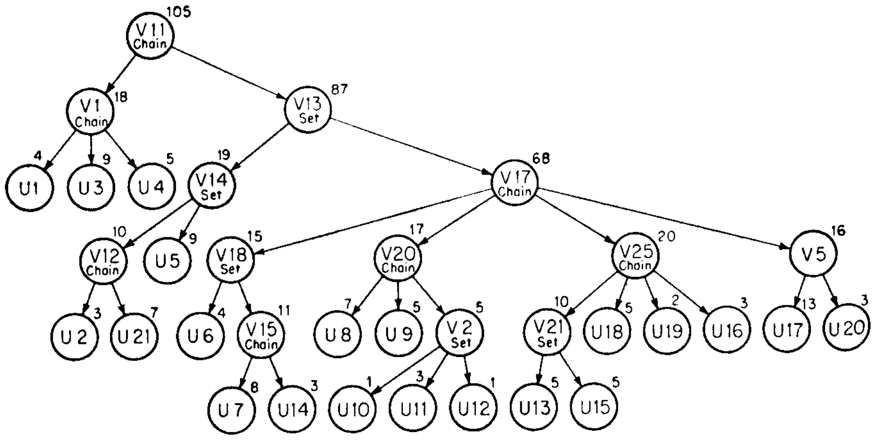


Figure 5. Twenty-one-element problem—Phase II output.

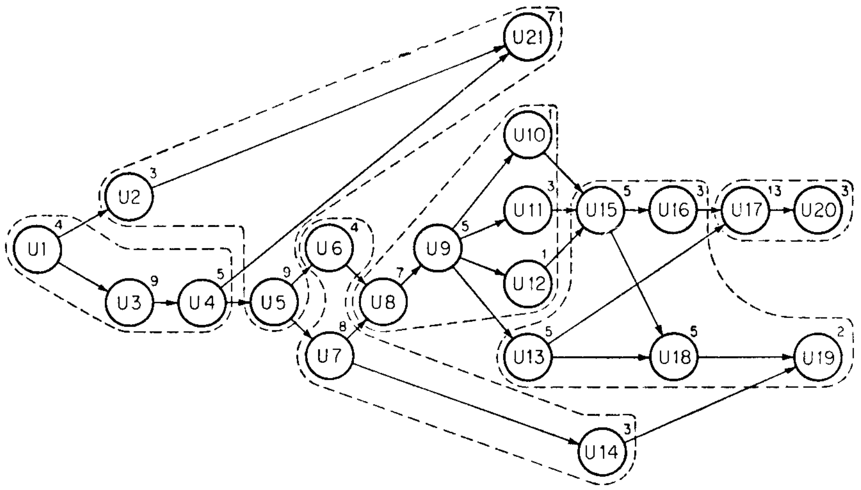


Figure 6. Twenty-one-element problem with stations, cycle time 20.

and the back, as well as which elements were left ungrouped in those solutions.

While these regrouping methods are clearly not foolproof, they have proven satisfactory for all problems attempted to date. Figure 5 shows the problem of Figs. 1 and 4 after the Phase II recursive routine has been applied to it (cycle time 20, 95 per cent usable time), and Fig. 6 pictures the work stations thus created on the original problem. (See also Table 1.) Appendix B lists the protocol produced during this assignment.

9. *Smoothing the Resulting Balance*

Phase III of the assembly line balancing program seeks to even the distribution of work among work stations by repeatedly reducing the time requirement of the largest work station. Inputs to this phase of the problem are a balanced line in hierarchical representation, such as would be produced by Phase II, and a cycle time. This iterative routine uses the same regrouping heuristics used by the Phase II recursive routine.

The following steps comprise the Phase III procedure:

1. Calculate the least possible time value of the highest station (with the given cycle time). If the largest station's time value is not greater than this bound, no further smoothing is possible and the routine terminates.

2. Given the largest station, consider in turn, in increasing order of size, all "adjacent" (in a set relationship or direct predecessors or followers) stations to that largest station. Try to even the distribution of work between

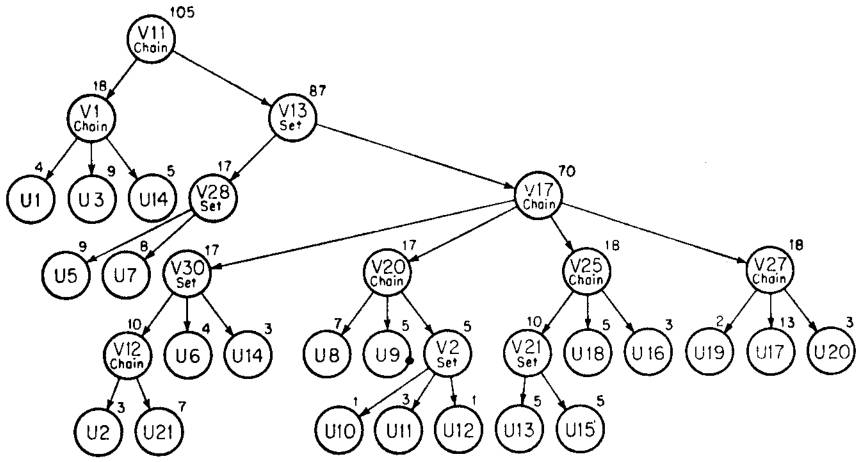


Figure 7. Twenty-one-element problem—Phase III output.

them using the direct transfer heuristic. If successful, return to step 1. If not, go to step 3.

3. Consider again those stations adjacent to the largest station in increasing order of size. Try to reduce the work load of the larger using the trading heuristic. If successful, return to step 1. If not, proceed to step 4.

4. Once again, consider adjacent stations in increasing order of size. For each one, using first direct transfer and then trading, try to make any transfer that reduces the largest station, even if the formerly smaller station is now as large or larger. If some such transfer is found, set up the subproblem of reducing this new larger station and apply this routine to it (excepting that step 4, setting up further subproblems, cannot be used). If successful, return to step 1; if not, the Phase III routine terminates.

The procedure outlined above has not yet been coded (which should be a relatively easy task, as it relies so heavily on already coded routines), but it has been hand-simulated for several cases. The results of this hand simulation for the problem of Figs. 1 and 4 is shown in Fig. 7. The simulated output of this phase of the assembly line balancing program is given in Appendix C.

10. Operating Results

Three sample problems were used in developing and testing this heuristic procedure. These are an 11-element problem taken from Jackson (1956), the 21-element problem used as an example here, taken from the Mitchell (1957), and a 70-element problem representing actual appliance industry

TABLE 2

	Phase I	Phase II					
Problem	Effort IPL instr.	Cycle time	Per cent usable	No. of stations	Per cent idle	Effort	Effort per station × 10^3
11-element	14,478	10	95	5	8	153,075	30.6
21-element	51,386	20	90	6	12	141,868	23.7
		19	95	6	8	143,183	23.8
		14	95	8	6	627,809	78.5
		18	98	6	3	483,458	80.5
		21	100	5	0	760,803	152.2
70-element	207,194	176	93	23	9	2,495,118	108.5

data. While these few cases do not completely test the method's general validity,[4] we can extract some interesting measures of performance in different types of problems.

Data summarizing these problem-solving exercises are given in Table 2. Note the increase in computing effort with problem-size in both Phases I and II. We also observe for the several cases of the 21-element problem an increase in effort per station as per cent idle time decreases. It appears that the amount of effort required to reach a balance depends upon the number of elements in the problem, number of stations desired and the per cent idle time available. Additional experience with actual problems will enable us to develop a method for estimating the effort required in particular cases.

To get some feeling for the effort required by exhaustive algorithms, we also attempted the 11-element problem with only the "exhaustive grouping" method, thus solving the problem using Jackson's algorithm (1956). (This is only a rough comparison, since the algorithm is carried out here using list processes and would benefit more than the heuristic program as a whole from use of matrix representation. Also, since the method is imbedded in the midst of our procedure, some amount of processing not required by that algorithm per se is included in the measure.) Under these conditions the program required about 389,000 IPL instructions to reach a balance, a factor of 2.5 over the heuristic approach. While this difference is not striking in light of the above warnings, the disparity will grow rapidly with increasing problem size.

We also tested Salveson's (1955) linear programming formulation of

[4] It is necessary to test this procedure against other large problems not only to measure its economic efficiency but also to ensure that the heuristics incorporated here have not been unconsciously adapted to meet the requirements of these particular test problems.

the assembly line balancing problem on our 11-element problem, using the RAND Simplex Code. The formulation required 11 equations in 62 unknowns (1 unknown for each possible legal grouping, given a cycle time of 10). This program required 15 iterations (starting from an artificial basis) to reach an optimum solution. As was expected, this answer contained groupings of fractional parts of elements. We must await an operating integer programming code in order to really test a linear programming approach.

The next areas we hope to analyze are (1) the effort saved through incremental changes—that is, starting from a nearby balance as opposed to starting from scratch with each new cycle time—and (2) the usefulness of initially producing balances at a range of interesting production rates, thus providing an approximation to the labor cost function for the assembly operation.

11. Adding the Zoning Constraint

One of the advantages sought from the heuristic approach to complex decision problems is the ability to redefine the problem, adding or deleting restrictions on a solution, with ease. As a specific example, we introduce the zoning restriction treated by Mitchell (1957). Under this restriction

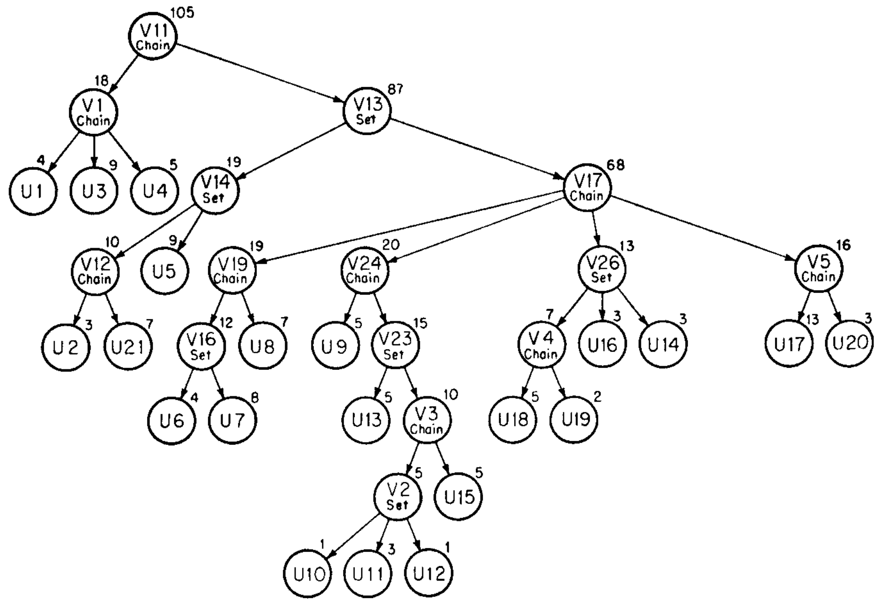


Figure 8. Twenty-one-element problem with zoning constraint—Phase II output.

TABLE 3

Zone	*Elemental tasks*
1	U1, U2, U3, U4, U5, U7, U8, U21
3	U3, U4, U6, U7, U8, U9, U10, U11, U12, U13, U15, U16, U18
4	U13, U14, U15, U16, U17, U18, U19, U20

each elemental task is assigned to one or more zones reflecting the physical limitations of the production process.

To incorporate this new restriction, the basic grouping generator (see *Regrouping Procedures*) is modified to produce only groupings within a specified zone or zones, and the routine that accepts work stations is modified to reject groups whose elements are not in a common zone. These modifications can be made relatively easily without affecting other parts of the over-all program. These modifications have been hand-simulated for several problems. Table 3 indicates the zoning of elements in the sample problem of Fig. 1. Figure 8 is the result of the modified Phase II operation on that problem (cycle time 20).

12. Mechanizing the Assembly Line Balancing Program

The assembly line balancing program described here is programmed in an interpretative system, IPL-IV, on the RAND Corporation's digital computer, JOHNNIAC. The IPL-IV system uses about 1200 of the JOHNNIAC's 4096 words of high-speed core memory and about 650 words of the 12,288-word auxiliary drum memory. In running the line balancing problem, which makes heavy use of the drum for temporary storage of data, the system interprets at the rate of about 9,000 to 10,000 IPL instructions per minute. While there has been no published account of IPL-IV, the closely related IPL-V for the IBM 650, 704 and 709 is fully documented (Newell and Tonge, 1960*c*, Newell et al., 1961*e*).

Why symbol manipulation languages[5] are extremely useful research tools for exploring techniques for problem-solving in complex and ill-structured situations is examined in more detail in another place (Tonge, 1961*a*). The essential point is that the "parameters" which the problem-solver wishes to vary freely are policies—general principles, rules of thumb, or what have you. The diversity of these principles cannot be anticipated when the problem-solving process begins with the problem of finding a good problem-solving process for the problem at hand. Some system for using the computer is required which frees the problem-solver from making

[5] Such programming languages include the IPL (Information Processing Language) Series (Newell and Tonge, 1960*c*; Newell et al., 1961*e*), LISP (McCarthy, 1959), COMIT (Yngve, 1958), and FORTRAN List Processing Language (Gelernter and Rochester, 1958).

early commitments (which he will later regret) about storage allocation, information to be carried, and so forth. This system must allow the user (possibly at the cost of machine speed and storage usage) to state his incompletely formulated problem and method of solution and start experimenting. List structure languages are a first step toward such a system.

The assembly line balancing program itself requires about 6700 machine locations, of which about 6100 are in auxiliary storage. In keeping with our preference for quick results rather than "efficient" operation, the entire structure of the problem was encoded in list form. With a satisfactory program (from the performance standpoint) developed, savings in machine space and speed could be made by encoding some structural information in matrix form. Bryton (1954), Marimont (1959), and others have made interesting contributions in this direction. Such a recoding would require only changing those routines that locate the value of certain properties of elements. The rest of the code is independent and would remain unchanged. Other reductions in operating time can be obtained by recoding in basic machine language certain operations now in interpretive code. Using IPL-V on the 704, it will be possible to implement the program described here at 5 times the speed achieved on the JOHNNIAC. At such speeds (about an hour maximum to balance an appliance line) this approach seems close to economic feasibility in treating some complex industrial decision problems.

13. Final Remarks

The aims of this research have been broader than just trying to produce a method for solving the assembly line balancing problem. We have set out to test the feasibility of a heuristic approach to complex decision-making in a particular industrial area and to examine the use of new computer techniques in implementing such an approach. As always, many interesting questions remain unexplored. But based on the experience reported here, we must conclude that the combination of a heuristic approach and these new methods of computer utilization are useful research tools in complex problem-solving.

Our program is not economically competitive when measured against the dollar per hour cost of line balancing by the industrial engineer. However, a true evaluation of the method must also consider (1) the possibility of fewer men required along the line on the average (a subject not yet fully explored), (2) the value of quick production of balances at a large number of production rates and (3) the value of releasing industrial engineers to do other, more creative analytic work.

Also, our research has indicated that present symbol manipulation languages, while a great advance for these purposes over conventional programming techniques, are still relatively primitive. While it is difficult to conceive preparing this program in a less powerful language than IPL-IV, much effort is required in machine-oriented aspects of communicating the problem-solving procedure to the computer. Further research in this area is needed before the heuristically oriented problem-solver will be free to expend his effort primarily on developing satisfactory problem-solving techniques, using the computer as a tool for spelling out the implications of his procedures.

Appendix A

Phase I—21-Element Problem

```
1 Propose chain U1
  2 Propose set U2, U3
    3 Propose chain U2
      4 Cannot reduce U2 to U21
    3 Propose chain U3
      4 Define chain V0 = U3, U4
      4 Propose set U5, U21
        5 Cannot reduce U21
    3 Propose Z V0, U2
      4 Propose chain U21
      4 Propose chain U5
        5 Propose set U6, U7
          6 Propose chain U6
            7 Cannot reduce U6 to U8
          6 Propose chain U7
            7 Propose set U8, U14
              8 Cannot reduce U8
          6 Propose Z U7, U6
            7 Delete constraint U14 to U19
            7 Propose chain U8
              8 Define chain V1 = U8, U9
              8 Propose set U10, U11, U12, U13
                9 Define set V2 = U10, U11, U12
                9 Propose chain V2
                  10 Define chain V3 = V2, U15
                  10 Propose set U16, U18
                     11 Cannot reduce U18
                9 Propose chain U13
                  10 Propose set U17, U18
                     11 Cannot reduce U17
                     11 Cannot reduce U18
                  9 Propose Z U13, V3
                    10 Delete constraint U13 to U17
                    10 Propose chain U18
                       11 Define chain V4 = U18, U19
                    10 Propose chain U16
                       11 Define chain V5 = U16, U17, U20
                    10 Define Z V6 = U13, V4, V3, V5
                8 Propose chain V6
                8 Define chain V7 = U8, U9, V6
              7 Propose chain U14
              7 Define Z V8 = U6, V7, U7, U14
          5 Propose chain V8
          5 Define chain V9 = U5, V8
        4 Define Z V10 = U2, U21, V0, V9
  2 Propose chain V10
  2 Define chain V11 = U1, V10
```

Appendix B

Phase II—21-Element Problem—Cycle Time 20—PCT Usable 90

```
1 Consider V11—components U1, V10—6 men available
  2 Activate direct transfer
    3 Combine V0, U1
      4 Redefine chain V11 = V1, V13
      4 Define chain V1 = U1, U3, U4
      4 Define chain V12 = U2, U21
      4 Define set V13 = V12, V9
  2 Assign 1 man to V1, 5 to V13
    3 Consider V1—components U1, U3, U4—1 man available
      4 Mark V1 as work station—time = 18
    3 Consider V13—components V12, V9—5 men available
      4 Assign 1 man to V12, 4 to V9
        5 Consider V12—components U2, U21—1 man available
          6 Combine U5, V12
            7 Redefine set V13 = V14, V8
            7 Define V14 = U5, V12
          6 Mark V14 as work station—time = 19
        5 Consider V8—components U6, V7, U7, U14
          6 Add U14, U6 to waiting list
          6 Assign 3 men to V7, 1 to U7
            7 Consider U7—1 man available
              8 Combine U7, U14, U6
                9 Define set V18 = U6, V15
      9 Define chain V17 = V18, V7
      9 Define chain V15 = U7, U14
      9 Redefine set V13 = V14, V17
    8 Mark V18 as work station—time = 15
  7 Consider V7—components U8, U9, V6—3 men available
    8 Activate sequential grouping
      9 Combine U8, U9, V2
        10 Redefine chain V7 = V20, V6
        10 Define V20 = U8, U9, V2
        10 Redefine Z V6 = U13, V4, U15, V5
    8 Assign 1 man to V20, 2 to V6
      9 Consider V20—components U8, U9, V2—1 man
        available
        10 Mark V20 as work station—time = 17
      9 Consider V6—components U13, V4, U15, V5—2
        men available
        10 Activate sequential grouping
           11 Combine U15, U13, V4, U16
              12 Define chain V23 = V25, V5
              12 Define chain V25 = V21, U18, U19,
                 U16
              12 Redefine chain V5 = U17, U20
              12 Define set V21 = U13, U15
              12 Redefine chain V7 = V20, V23
        10 Assign 1 man to V25, 1 to V5
           11 Consider V5—components U17, U20—1
              man available
              12 Mark V5 as work station—time = 16
           11 Consider V25—components V21, U18,
              U19, U16—1 man available
              12 Mark V25 as work station—time = 20
  7 Define chain V17 = V18, V20, V25, V5
```

Appendix C

Phase III—21-Element Problem—Cycle Time 20—PCT Usable 90

```
1 Least possible high station = 105/6 = 17+ = 18 time units
1 Consider V25—time = 20
  2 Activate direct transfer, V25 to V5
    3 Combine U19, V5
      4 Redefine chain V17 = V18, V20, V25, V27
      4 Redefine chain V25 = V21, U18, U16
      4 Define chain V27 = U19, U17, U20
```

```
1 Consider V14—Time = 19
  2 Activate direct transfer, V14 to V18
  2 Activate direct transfer, V14 to V1
  2 Activate trade, V14 and V18
    3 Combine V14, U7
      4 Redefine set V13 =V28, V17
      4 Redefine set V28 = V12, U5, U7
      4 Redefine set V18 = U6, U14
    3 Combine V12, V18
      4 Redefine set V28 = U5, U7
      4 Redefine chain V17 = V30, V20, V25, V27
      4 Define set V30 = V12, U6, U14
```

A HEURISTIC PROGRAM THAT SOLVES SYMBOLIC INTEGRATION PROBLEMS IN FRESHMAN CALCULUS

by James R. Slagle

A large high-speed general-purpose digital computer (IBM 7090) was programmed to solve elementary symbolic integration problems at approximately the level of a good college freshman. The program is called SAINT, an acronym for "Symbolic Automatic INTegrator." The SAINT program is written in LISP (McCarthy, 1960), and most of the work reported here is the substance of a doctoral dissertation at the Massachusetts Institute of Technology (Slagle, 1961). This discussion concerns the SAINT program and its performance.

Some typical samples of SAINT's external behavior are given so that the reader may think in concrete terms. Let SAINT read in its card reader an IBM card containing (in a suitable notation) the symbolic integration problem $\int xe^{x^2}\,dx$. In less than a minute and a half, SAINT prints out the answer, $\frac{1}{2}e^{x^2}$. Except where otherwise noted, every problem mentioned in this chapter has been solved by SAINT. Note that SAINT omits the constant of integration, and we, too, shall ignore it throughout our discussion. After working for less than a minute on the problem $\int e^{x^2}\,dx$ (which cannot be integrated in elementary form) SAINT prints out that it cannot solve it.

SAINT performs indefinite integration, also called antidifferentiation. In addition it performs definite and multiple integration when these are trivial extensions of indefinite integration. SAINT handles integrands that represent explicit elementary functions of a real variable which, for the sake of brevity, will be elementary functions. The elementary functions are the functions normally encountered in freshman integral calculus, except that SAINT does not handle hyperbolic notation. The elementary functions are defined recursively as follows:

a. Any constant is an elementary function.
b. The variable is an elementary function.
c. The sum or product of elementary functions is an elementary function.
d. An elementary function raised to an elementary function power is an elementary function.
e. A trigonometric function of an elementary function is an elementary function.
f. A logarithmic or inverse trigonometric function of an elementary function (restricted in range if necessary) is an elementary function.

Currently SAINT uses twenty-six standard forms. It uses eighteen kinds of transformations including integration by parts and various substitution methods (but excluding, among others, the method of partial fractions). Since the SAINT program uses heuristic methods, it is by definition a heuristic program. Although many authors have given many definitions, in this discussion a heuristic method (or simply a heuristic) is a method which helps in discovering a problem's solution by making plausible but fallible guesses as to what is the best thing to do next.

Indefinite Integration Procedure of SAINT

This section describes how SAINT performs indefinite integration. An attempt is made to orient the reader before a detailed description of the procedure is given. The executive organization of SAINT is like that of the Logic Theorist of Newell, Shaw, and Simon (1957). It will help to take a preview of Sec. 14 (especially Fig. 3). The "try for an immediate solution" mentioned twice in Fig. 3 may be described roughly as follows: As soon as a new goal *g* is generated, SAINT uses its straightforward methods in an attempt to achieve it. While doing this, SAINT may add *g* or certain of *g*'s subgoals to the "temporary goal list." If *g* is achieved, an attempt is made to achieve the original goal. Slagle (1961) includes among other things, a full description together with a detailed example and suggestions for future work.

As a concrete example we sketch how SAINT solved

$$\int \frac{x^4}{(1 - x^2)^{5/2}}\,dx$$

in eleven minutes. SAINT's only guess at a first step is to try substitution: $y = \arcsin x$, which transforms the original problem into

$$\int \frac{\sin^4 y}{\cos^4 y}\,dy$$

For the second step SAINT makes three alternative guesses:

A. By trigonometric identities $\int \frac{\sin^4 y}{\cos^4 y}\,dy = \int \tan^4 y\,dy$

B. By trigonometric identities $\int \frac{\sin^4 y}{\cos^4 y}\,dy = \int \cot^{-4} y\,dy$

C. By substituting $z = \tan(y/z)$ $\int \frac{\sin^4 y}{\cos^4 y}\,dy = \int 32 \frac{z^4}{(1+z^2)(1-z^2)^4}\,dz$

SAINT immediately brings the 32 outside of the integral.

After judging that (*A*) is the easiest of these three problems SAINT guesses the substitution $z = \tan y$, which yields

$$\int \tan^4 y\,dy = \int \frac{z^4}{1+z^2}\,dz$$

SAINT immediately transforms this into

$$\int \left(-1 + z^2 + \frac{1}{1+z^2}\right) dz = -z + \frac{z^3}{3} + \int \frac{dz}{1+z^2}$$

Judging incorrectly that (*B*) is easier than

$$\int \frac{dz}{1+z^2}$$

SAINT temporarily abandons the latter and goes off on the following tangent. By substituting $z = \cot y$,

$$\int \cot^{-4} y\,dy = \int -\frac{dz}{z^4(1+z^2)} = -\int \frac{dz}{z^4(1+z^2)}$$

Now SAINT judges that

$$\int \frac{dz}{1+z^2}$$

is easy and guesses the substitution, $w = \arctan z$ which yields $\int dw$. Immediately SAINT integrates this, substitutes back and solves the original problem.

$$\int \frac{x^4}{(1-x^2)^{5/2}}\,dx = \arcsin x + \tfrac{1}{3} \tan^3 \arcsin x - \tan \arcsin x$$

The indefinite integration procedure may be described as follows:

1. Goals

In each application of the present procedure, the solutions of certain problems, namely, performing integrations with side conditions, are goals. How goals are generated, manipulated, and achieved is described later. For now, let us limit ourselves to describing what we shall call the "original goal," which consists of the originally given integrand and variable of integration.

2. The Goal List

The original goal is made the first member of a list called the goal list. From time to time new goals may be generated. Each newly generated goal is added to the end of the goal list.

3. Standard Forms

Whenever an integrand of a newly generated goal is of "standard form," that goal is immediately achieved by substitution. An integrand is said to be of standard form when it is a substitution instance of one of a certain set of forms. For example, $\int 2^x\,dx$ is an instance of $\int c^v\,dv = c^v/(\ln c)$ and hence has the solution $2^x/(\ln 2)$. Currently SAINT uses twenty-six standard forms (Slagle, 1961).

4. Algorithmlike Transformations

Whenever an integrand is found to be not of standard form, it is tested to see if it is amenable to an algorithmlike transformation. By an algorithmlike transformation is meant a transformation which, when applicable, is always or almost always appropriate. For a goal, a transformation is called appropriate if it is the correct next step to bring that goal nearer to achievement. Three of the eight algorithmlike transformations used in SAINT are:

a. Factor constant, *i.e.*,

$$\int cg(v)\,dv = c\int g(v)\,dv$$

b. Decompose, *i.e.*,

$$\int \Sigma g_i(v)\,dv = \Sigma \int g_i(v)\,dv$$

c. Linear substitution, *i.e.*, if the integral is of the form

$$\int f(c_1 + c_2 v)\,dv$$

substitute $u = c_1 + c_2 v$, and obtain an integral of the form

$$\int \frac{1}{c_2} f(u)\,du$$

for example, in

$$\int \frac{\cos 3x}{(1 - \sin 3x)^2}\,dx$$

substitute $y = 3x$.

5. The AND-OR Goal Tree

When a heuristic transformation (to be described in Sec. 11) or an algorithmlike transformation is applied to a goal, new goals are generated. These goals, in turn, may generate more goals, and a certain hierarchy is created. Such a hierarchy is conveniently represented by a graph or tree growing downward. To facilitate understanding, the terminology of ordinary and family trees is adopted by analogy, for example pruning, alive, dead, child, parent, descendant, ancestor, etc.

Suppose we have an integration to perform, or more generally, any goal *g*, which we shall represent graphically by a point. A goal may be transformed into one or more subgoals which may be related to the goal in many ways. This integration procedure incorporates two common relations, namely AND and OR.

a. AND relationship

An AND relationship between a goal and at least two subgoals exists when the achieving of all of the subgoals causes the achieving of the goal. Figure 1 depicts a relationship with three subgoals. The arc joining the three branches denotes the AND relationship.

b. OR relationship

An OR relationship between a goal and its subgoals exists when the achieving of any one of the subgoals causes the achieving of the goal. Examples of this will appear later.

From these two basic relationships, more complicated relationships among goals may be built up; for example, see Fig. 2*a* and *b* in Sec. 12.

6. The Temporary Goal List

The first attempt on new goals is performed by the procedures "imsln" ("IMmediate SoLutioN") described in Sec. 13 below. Any goal en-

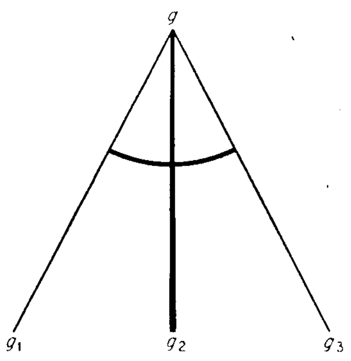


Figure 1. An AND relationship.

countered by imsln which is neither of standard form nor amenable to an algorithmlike transformation is added to the end of the "temporary goal list" (not to be confused with the "goal list") and later transferred to the "heuristic goal list" described in Sec. 10 below. If the goal were added directly to the heuristic goal list rather than to the temporary goal list, time might be wasted by finding the goal's character (cf. Sec. 8).

7. The Role of the Resource Allotment

The resource allotment, or the total amount of work space, is a side condition of the original goal. Before proceeding to apply heuristic transformations, it must be verified that the resource allotment has not been exceeded. If the resource allotment has been exceeded, SAINT reports this fact as its final answer. Although other kinds of resources, for example, time, might also be considered, the only kind of resource allotment handled by SAINT is the total amount of work space. For hand simulation, the work space can be measured by the number of pages or by the number of lines used for the final and all intermediate results.

8. Character of a Goal

When a goal is taken off the temporary goal list its "character" is obtained, that is, an ordered list of "characteristics." A characteristic of a goal is a feature which might be useful either in estimating the cost of attempting its attainment or in selecting appropriate heuristic transformations (see Sec. 11). In SAINT, the character is composed of eleven characteristics of the integrand (Slagle, 1961) including its function type (whether it is a rational function, algebraic function, rational function of sines and cosines, etc.) and its depth. The depth of an integrand is the maximum level of function composition which occurs in that expression:

$$\begin{aligned} x &\text{ is of depth } 0,\\ x^2 &\text{ is of depth } 1,\\ e^{x^2} &\text{ is of depth } 2,\\ xe^{x^2} &\text{ is of depth } 3. \end{aligned}$$

As one might guess, this helps get a crude estimate of the problem's difficulty.

9. Relative Cost Estimate

Although other estimates could be tried, for the relative cost estimate of a goal we take simply the depth of its integrand. This makes use of the fact that, ordinarily, the deeper the integrand the more will be the resources needed to investigate that goal.

10. The Heuristic Goal List

A list of goals requiring heuristic transformations, or, more briefly, a heuristic goal list, is an ordered list of those goals which are neither of standard form nor amenable to an algorithmlike transformation. A member of the heuristic goal list is called a heuristic goal. New such goals are inserted in order of increasing relative cost estimate.

11. The Heuristic Transformations

A transformation of a goal is called heuristic when, even though it is applicable and plausible, there is a significant risk that it is not the appropriate next step. A transformation may be inappropriate either because it leads no closer to the solution or because some other transformation would be better. The heuristic transformations are analogous to the methods of detachment, forward chaining and backward chaining in the Logic Theorist of Newell, Shaw, and Simon (1957). The ten types of heuristic transformation (Slagle, 1961) used by SAINT are designed to suggest plausible transformations of the integrand, substitutions, and attempts using the method of integration by parts. Below is given only the most successful heuristic, "substitution for a subexpression whose derivative divides the integrand."

Let $g(v)$ be the integrand. For each nonconstant nonlinear subexpression $s(v)$ such that neither its main connective is MINUS nor is it a product with a constant factor, and such that the number of nonconstant factors of the fraction $g(v)/s'(v)$ (after cancellation) is less than the number of factors of $g(v)$, try substituting $u = s(v)$. Thus, in $xe^{x^2}\,dx$, substitute $u = x^2$. (When SAINT tried this problem it used this heuristic but surprised me by substituting $u = e^{x^2}$, which is somewhat better.)

12. Pruning the Goal Tree

Whenever some goal g has been achieved, the goal tree is pruned, that is, certain closely related goals are automatically achieved and certain other goals, newly rendered superfluous, are killed.

The pruning procedure will be clarified by an example. In Fig. 2*a* the achieving of g_{221} allows g_{22} to be achieved (since, as indicated by the black dot, g_{222} has already been achieved). In turn, the achieving of g_{22} allows g_2 to be achieved (since there is an OR relationship). Since the achieving of g_2 now has rendered g_{23} superfluous, it is killed. However, another of g_2's children g_{12} is not killed since, through its other parent g_1 it has direct living ancestry to the original goal g. The original goal g cannot be achieved from the achieving of g_2 since there is an AND relationship and g_1 has not yet been achieved. Therefore, the result of the pruning process is as shown

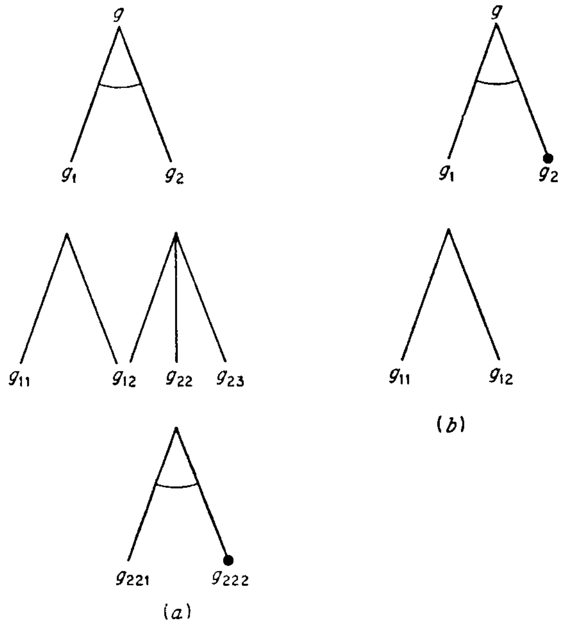


Figure 2.

in Fig. 2*b*. If either g_{11} or g_{12} is later achieved, the original goal could and would be achieved.

13. Trying for an Immediate Solution

As soon as a new goal g is generated, SAINT uses its straightforward methods in an attempt to achieve it. While doing this, SAINT may add g or certain of g's subgoals to the temporary goal list. If g is achieved, an attempt is made to achieve the original goal.

14. Executive Organization

Precisely how all the various elements 1 through 13 are pieced together to form an integration procedure is described below. The original goal is given as a triplet, namely, the integrand, the variable of integration, and the resource allotment. The procedure (see Fig. 3) is as follows:

a. If a try for an immediate solution with the original goal is successful, return with the answer, the actual indefinite integral.

b. If the resource allotment has been exceeded, report failure.

c. Obtain and associate with each goal on the temporary goal list its character and relative cost estimate. Take the goals off the temporary goal list, and insert each one in the heuristic goal list according to its relative cost estimate. If no goals remain on the heuristic goal list, report failure.

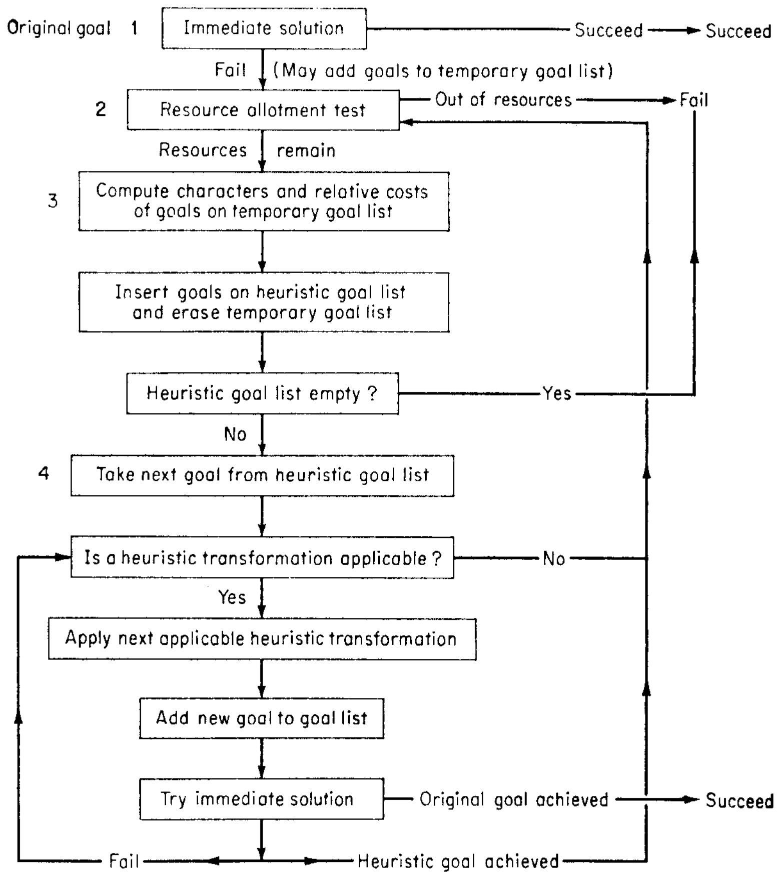

Original goal
1
Immediate solution
Succeed
Succeed
Fail
(May add goals to temporary goal list)
2
Resource allotment test
Out of resources
Fail
Resources remain
3
Compute characters and relative costs of goals on temporary goal list
Insert goals on heuristic goal list and erase temporary goal list
Heuristic goal list empty ?
Yes
No
4
Take next goal from heuristic goal list
5
Is a heuristic transformation applicable ?
No
Yes
6
Apply next applicable heuristic transformation
Add new goal to goal list
Try immediate solution
Original goal achieved
Succeed
Fail
Heuristic goal achieved

Figure 3.

d. Take the next goal g_i off the heuristic goal list and let it be the goal under consideration in the following inner loop.

e. If no heuristic transformations applicable to g_i remain, go to step *b*.

f. Apply the next heuristic transformation applicable to g_i. As soon as a new goal g is so generated, add it to the goal list, and try for an immediate solution with g. Then there are three cases. If this try achieves the original goal, return with the answer. Failing this, if g_i is achieved, go to step *b*. Otherwise go to step *e*.

Definite Integration Procedure

SAINT can perform some definite integrations by first finding the corresponding indefinite integrals. Thus, for example, for the problem

$$\int_0^3 x\sqrt{x^2+16}\,dx$$

SAINT first finds the indefinite integral,

$$\int x\sqrt{x^2+16}\,dx = \tfrac{1}{3}(x^2+16)^{3/2}$$

SAINT substitutes the limits and obtains the answer $^{6}\tfrac{1}{3}$.

Multiple Integration Procedure

SAINT can perform multiple integration when it can perform the required definite integrations, *e.g.*,

$$\int_1^1 \int_{y^2}^{2-y^2} dx\,dy$$

Experiments and Findings with SAINT

This section describes some of SAINT's typical observed behavior and how one modification changes its behavior. Slagle (1961) describes other experiments and gives further details. The experiments to measure SAINT's behavior involve 86 problems. Largely for the purposes of debugging, 32 of the problems were selected or constructed by the author, who fully expected SAINT to solve them all. More objectively, the remaining 54 problems were selected from MIT freshman calculus final examinations by the author's assistant, Gerald Shapiro, with instructions to select the more diverse and difficult problems, provided only that the method of partial fractions was not needed for the solution. The measures of behavior that we use are:

1. Power
The power of a version of SAINT refers to the size of the class of problems that it can solve.
2. Time
All the times mentioned refer to the IBM 7090 computer.
3. Number of subgoals and unused subgoals
The original goal is not included in the number of subgoals. An unused subgoal is a subgoal which is not needed in the solution chain.
4. Level
The level of a solution is the maximum level at which a used subgoal occurs in the goal tree during that solution.
5. Heuristic level of a solution
This measure is similar to "level" except that only the goal-tree branches representing heuristic transformations are considered rather than all the branches representing algorithmlike or heuristic transformations.

A. Unmodified SAINT

The SAINT program described in the preceding sections tried to solve all 86 problems selected by the author and Gerald Shapiro. In this attempt, the computer spent about half of its time in reclaiming abandoned memory registers for reuse. Approximately half of the remaining time was spent in pattern recognition, that is, in finding characters and in recognizing when an integrand is of standard form or amenable to an algorithmlike or heuristic transformation. As the author expected, SAINT solved all 32 of his problems. Of the 54 MIT problems, SAINT solved 52 and quickly (in less than a minute) reported failure for the other 2. Both of the failures are excluded from the averages in the table below, which summarizes SAINT's average performance.

SAINT's Average Performance

	Minutes	Subgoals	Unused subgoals	Level	Heuristic level
32 author problems	3.3	6.4	2.0	3.5	1.0
52 MIT problems	2.0	4.7	0.8	2.9	0.8
All 84 problems	2.4	5.3	1.25	3.0	0.9

In this paragraph we complement the tabulation of SAINT's average performance on MIT problems with some examples of SAINT's extreme behavior. For this purpose, only MIT problems are considered since they were selected more objectively. SAINT seemed to find $\int_1^e (dx/x)$ the easiest problem since it generated no subgoals at all and took the least time, namely 0.03 minute. SAINT took the most time, 18 minutes, for

$$\int \frac{\sec^2 t}{1 + \sec^2 t - 3 \tan t}\, dt$$

whose solution ties for the maximum heuristic level of four. The other problem whose SAINT solution has a heuristic level of four is

$$\int \frac{x^4}{(1 - x^2)^{5/2}}\, dx \quad .$$

The maximum heuristic level obtained by the unmodified Logic Theorist is two, which occurred for two of 38 solutions (Newell, Shaw, and Simon, 1957*a*). SAINT generated the most subgoals (18) and had the maximum level (8) for $(\sin x + \cos x)^2\, dx$. In 37 of the 52 problems, SAINT generated only subgoals that were needed in the solution chain. In this regard, SAINT registered its best performance on one of these 37 problems

$$\int_{5/2}^{4} \int_{x(11-2x)}^{4(x-1)(5-x)} dy\, dx$$

for which SAINT generated 16 subgoals, all of which were needed in the solution chain.

B. BSAINT, *i.e.*, SAINT Trying Heuristic Goals in Order of Generation

Instead of trying heuristic goals in order of increasing depth, BSAINT tries heuristic goals merely in the order in which they were generated. In measures of performance including time and the number of unused subgoals, SAINT was better than BSAINT in three of the four problems which caused a difference in behavior.

Main Conclusions

The conclusions are based on experience, namely, the experiments described in the preceding sections and the author's experience concerning the creation, structure, and performance of SAINT. Throughout this section, a parenthesized mention of an experiment is an appeal for support of a conclusion to an experiment described in the previous section.

1. A machine can manifest intelligent problem-solving behavior, that is, behavior which, if performed by people, would be called intelligent (experiment *A*).
2. A heuristic program can easily include programs for handling an AND-OR goal tree (such as found in SAINT), which is often useful in complex goal-achieving schemes.
3. In SAINT, pattern recognition plays a very important part in three senses.
   a. Pattern recognition consumes much of the program and programming effort.
   b. Pattern recognition is used frequently and with great variety, for instance, in determinations involving standard forms, algorithmlike and heuristic transformations, and relative cost estimates.
   c. Pattern recognition consumes much of the time in solving integration problems (experiment *A*).
4. The tripartite division of methods into standard forms, algorithmlike transformations, and heuristic transformations is very useful in problem-solving. Standard forms in SAINT and "substitution" in the Logic Theorist may be instances of an "immediately achieve" procedure which seems to be a basic component of a goal-achieving scheme. The input to the procedure is a goal. The output is "no" (the goal is not yet achieved) or one or more of the following three items, namely, "yes," how to achieve the goal, or the achievement of the goal. In each domain, the procedure for immediately achieving a goal must be supplied anew and, since it is a very frequently used procedure, should operate very rapidly. The algorithmlike transformations also seem to be a basic component of a goal-achieving scheme, but this remains to be seen since they are not present in all schemes, for ex-

ample, the Logic Theorist. The organization of SAINT's heuristic transformations (corresponding to that of the Logic Theorist's methods of detachment, forward chaining, and backward chaining) seems to be an often convenient but not a basic component of a goal-achieving scheme.

5. A fourfold increase in SAINT's memory size (now 32,768 registers) could have been readily converted into a hundredfold increase in speed, since the reclaiming of abandoned memory registers for reuse, which now accounts for about half of the running time, would become insignificant and since a compiled program would run about fifty times faster. Much computer time and space could be saved if one computer instruction represented the very frequently used symbol manipulating functions.

6. The present speed of SAINT compares very favorably with the speed of the average college freshman (experiment *A*). With a now commercially available large high-speed digital computer, such as the IBM 7030 (STRETCH), a compiled but otherwise unimproved SAINT program would run eight hundred times faster, which would far surpass in speed even the most gifted of mathematicians at this task. At present commercial rates, an IBM 7090 SAINT solution of an average MIT final-examination problem costs about fifteen dollars, far more expensive than a human solution. However, a STRETCH SAINT solution would cost only about two dollars or, if compiled, only about four cents. This rapidly decreasing cost trend in computers, not to mention possible improvements in the SAINT program, will result in solutions which are far cheaper by machine than by man.

section 5

Question-answering Machines

Question-answering machines are computer programs that can be interrogated in natural language (with some constraints) for the answers to questions about a universe of discourse. The problem of constructing such machines has both theoretical and practical interest.

From the theoretical point of view, one of the most intriguing pursuits in artificial intelligence research is that of discovering the information processing structure underlying the act of "comprehending" or the process of "understanding." Though some will view the use of these two words with alarm, we use them here invested with their full human meaning. For this is precisely the goal of the researchers themselves. They are engaged in a quest for an information processing theory of an intelligent mechanism capable of "grasping the meaning" of those strings of symbols which are our natural language sentences and questions.

As a practical matter, throughout the entire field of computer applications—in information retrieval applications, in computer-controlled libraries and classrooms, in military command-control applications, in the research toward man-machine cooperative problem-solving, to mention just a few areas—users are feeling acutely the need to communicate with computers more directly, with greater fluidity and facility, than has been possible in the past.

The two reports reprinted in this section describe important initial steps toward these goals. The BASEBALL program of Green, Wolf, Chomsky, and Laughery is, in sportswriters' parlance, a baseball buff, answering certain kinds of queries about baseball "facts and figures" from a stylized baseball "yearbook" stored in the computer's memory. Lindsay's SAD SAM program answers questions about family kinship relations.

The basic differences in approach between the two efforts are as follows:

1. In the BASEBALL project, the "fact library" is stored in the computer memory in advance in a form which makes searching the library relatively easy. In SAD SAM, the kinship relationships between members of the particular family under discussion are not given the program in advance but are input to the program in the form of English sentences. The program, assimilating and analyzing these sentences, constructs an internal "map," or "model," of the familial relationships—an information structure referenced later in answering questions about the family.

2. In BASEBALL, within certain constraints upon the type of question that can be asked and the availability of words in the program's glossary, full-blown English is the language of discourse. In SAD SAM a simplified English (the so-called Basic English system, devised to facilitate English language learning) is the language of discourse.

Both programs, of course, operate within extremely limited fact universes. At this exploratory stage of the research, however, discovery and thorough understanding of possible basic mechanisms for language comprehension would appear to be more important than premature attempts to deal with wide-ranging universes of discourse.

The BASEBALL research was done at Lincoln Laboratory under the direction of Bert Green, who is presently Chairman of the Psychology Department at the Carnegie Institute of Technology.

Alice Wolf is presently a member of the research staff of Bolt, Beranek, and Newman in Cambridge, Massachusetts.

Carol Chomsky is associated with Harvard University.

Kenneth Laughery is a member of the faculty of the University of Buffalo.

Robert Lindsay is a member of the faculty of the Psychology Department, and a member of the staff of the Computer Center, at the University of Texas.

BASEBALL: AN AUTOMATIC QUESTION ANSWERER

by Bert F. Green, Jr., Alice K. Wolf, Carol Chomsky, & Kenneth Laughery

Introduction

Men typically communicate with computers in a variety of artificial, stylized, unambiguous languages that are better adapted to the machine than to the man. For convenience and speed, many future computer-centered systems will require men to communicate with computers in natural language. The business executive, the military commander, and the scientist need to ask questions of the computer in ordinary English, and to have the computer answer the questions directly. Baseball is a first step toward this goal.

Baseball is a computer program that answers questions posed in ordinary English about data in its store. The program consists of two parts. The linguistic part reads the question from a punched card, analyzes it syntactically, and determines what information is given about the data being requested. The processor searches through the data for the appropriate information, processes the results of the search, and prints the answer.

The program is written in IPL-V (Newell, et al., 1960*e*), an information processing language that uses lists, and hierarchies of lists, called list structures, to represent information. Both the data and the dictionary are list structures, in which items of information are expressed as attribute-value pairs, *e.g.*, Team = Red Sox.

The program operates in the context of baseball data. At present, the data are the month, day, place, teams and scores for each game in the American League for one year. In this limited context, a small vocabulary is sufficient, the data are simple, and the subject matter is familiar.

Some temporary restrictions were placed on the input questions so that the initial program could be relatively straightforward. Questions are limited to a single clause; by prohibiting structures with dependent clauses the syntactic analysis is considerably simplified. Logical connectives, such as *and, or,* and *not,* are prohibited, as are constructions implying relations like *most* and *highest.* Finally, questions involving sequential facts, such as "Did the Red Sox ever win six games *in a row?*" are prohibited. These restrictions are temporary expedients that will be removed in later versions of the program. Moreover, they do not seriously reduce the number of questions that the program is capable of answering. From simple questions such as "Who did the Red Sox lose to on July 5?" to complex questions such as "Did every team play at least once in each park in each month?" lies a vast number of answerable questions.

Specification List

Fundamental to the operation of the baseball program is the concept of the *specification list,* or *spec list.* This list can be viewed as a canonical expression for the meaning of the question; it represents the information contained in the question in the form of attribute-value pairs, *e.g.,* Team = Red Sox. The spec list is generated from the question by the linguistic part of the program, and it governs the operation of the processor. For example, the question "Where did the Red Sox play on July 7?" has the spec list:

Place = ?
Team = Red Sox
Month = July
Day = 7.

Some questions cannot be expressed solely in terms of the main attributes (Month, Day, Place, Team, Score, and Game Serial Number), but require some modification of these attributes. For example, on the spec list of "What teams won 10 games in July?", the attribute Team is modified by Winning, and Game is modified by Number of, yielding

$$\begin{aligned}\text{Team}_{\text{(winning)}} &= ?\\ \text{Game}_{\text{(number of)}} &= 10\\ \text{Month} &= \text{July.}\end{aligned}$$

Dictionary

The dictionary definitions, which are expressed as attribute-value pairs, are used by the linguistic part of the program in generating the spec list. A complete definition for a word or idiom includes a part of speech, for use in determining phrase structure; a meaning, for use in analyzing content; an indication of whether the entry is a question word, *e.g., who* or *how many;* and an indication of whether a word occurs as part of any stored idiom. Separate dictionaries are kept for words and idioms, an idiom being any contiguous set of words that functions as a unit, having a unique definition.

The meaning of a word can take one of several forms. It may be a main or derived attribute with an associated value. For example, the meaning of the word *Team* is Team = (blank), the meaning of *Red Sox* is Team = Red Sox, and the meaning of *who* is Team = ?. The meaning may designate a subroutine, together with a particular value, as in the case of modifiers such as *winning, any, six,* or *how many.* For example, *winning* has the meaning Subroutine A1 = Winning. The subroutine, which is executed by the content analysis, attaches the modifier Winning to the attribute of the appropriate noun. Some words have more than one meaning; the word *Boston* may mean either Place = Boston or Team = Red Sox. The dictionary entry for such words contains, in addition to each meaning, the designation of a subroutine that selects the appropriate meaning according to the context in which the word is encountered. Finally, some words such as *the, did, play,* etc., have no meaning.

Data

The data are organized in a hierarchical structure, like an outline, with each level containing one or more items of information. Relationships among items are expressed by their occurrence on the same list, or on associated lists. The main heading, or highest level of the structure, is the attribute Month. For each month, the data are further subdivided by place. Below each place under each month is a list of all games played at that place during that month. The complete set of items for one game is found by tracing one path through the hierarchy, *i.e.* one list at each level. Each path contains values for each of six attributes, *e.g.:*

```
Month = July
  Place = Boston
    Day              = 7
    Game Serial No.  = 96
    (Team = Red Sox, Score = 5)
    (Team = Yankees, Score = 3)
```

The parentheses indicate that each Team must be associated with its own score, which is done by placing them together on a sublist.

The processing routines are written to accept any organization of the data. In fact, they will accept a nonparallel organization in which, for example, the data might be as above for all games through July 31, and then organized by place, with month under place, for the rest of the season. The processing routines will also accept a one-level structure in which each game is a list of all attribute-value pairs for that game. The possibility of hierarchical organization was included for generality and potential efficiency. The basic rule is that any one path through the data, including one list at each level, must contain all of the facts for a single game. Also, on every such path, each attribute may occur at most once, unless it occurs on parallel sublists.

Details of the Program

The program is organized into several successive, essentially independent routines, each operating on the output of its predecessor and producing an input for the routine that follows. The linguistic routines include question read-in, dictionary look-up, syntactic analysis, and content analysis. The processing routines include the processor and the responder.

Linguistic Routines

QUESTION READ-IN

A question for the program is read into the computer from punched cards. The question is formed into a sequential list of words.

DICTIONARY LOOK-UP

Each word on the question list is looked up in the word dictionary and its definition copied. Any undefined words are printed out. (In the future, with a direct-entry keyboard, the computer can ask the questioner to define the unknown words in terms of words that it knows, and so augment its vocabulary.) The list is scanned for possible idioms; any contiguous words that form an idiom are replaced by a single entry on the question list, and an associated definition from the idiom dictionary. At this point, each entry on the list has associated with it a definition, including a part of speech, a meaning, and perhaps other indicators.

SYNTAX

The syntactic analysis is based on the parts of speech, which are syntactic categories assigned to words for use by the syntax routine. There are 14 parts of speech and several ambiguity markers.

First, the question is scanned for ambiguities in part of speech, which are resolved in some cases by looking at the adjoining words, and in other cases by inspecting the entire question. For example, the word *score* may be either a noun or a verb; our rule is that, if there is no other main verb in the question, then *score* is a verb, otherwise it is a noun.

Next, the syntactic routine locates and brackets the noun phrases, [□] and the prepositional and adverbial phrases, (□). The verb is left unbracketed. This routine is patterned after the work of Harris and his associates at the University of Pennsylvania (Harris, 1960). Bracketing proceeds from the end of the question to the beginning. Noun phrases, for example, are bracketed in the following manner: certain parts of speech indicate the end of a noun phrase; within a noun phrase, a part of speech may indicate that the word is within the phrase, or that the word starts the phrase, or that the word is not in the phrase, which means that the previous word started the phrase. Prepositional phrases consist of a preposition immediately preceding a noun phrase. The entire sequence, preposition and noun phrase, is enclosed in prepositional brackets. An example of a bracketed question is shown below:

[How many games] did [the Yankees] play (in [July])?

When the question has been bracketed, any unbracketed preposition is attached to the first noun phrase in the sentence, and prepositional brackets added. For example, "Who did the Red Sox lose to on July 5?" becomes "(To [who]) did [the Red Sox] lose (on [July 5])?"

Following the phrase analysis, the syntax routine determines whether the verb is active or passive and locates its subject and object. Specifically, the verb is passive if and only if the last verb element in the question is a main verb and the preceding verb element is some form of the verb *to be.* For questions with active verbs, if a free noun phrase (one not enclosed in prepositional brackets) is found between two verb elements, it is marked *Subject,* and the first free noun phrase in the question is marked *Object.* Otherwise the first free noun phrase is the subject, the next, if any, is the object. For passive verbs, the first free noun phrase is marked *Object* (since it is the object in the active form of the question) and all prepositional phrases with the preposition *by* have the noun phrase within them marked *Subject.* If there is more than one, the content analysis later chooses among them on the basis of meaning.

Finally, the syntactic analysis checks to see if any of the words is marked as a question word. If not, a signal is set to indicate that the question requires a *yes/no* answer.

CONTENT ANALYSIS

The content analysis uses the dictionary meanings and the results of the syntactic analysis to set up a specification list for the processing program. First any subroutine found in the meaning of any word or idiom in the question is executed. The subroutines are of two basic types; those that deal with the meaning of the word itself and those that in some way change the meaning of another word. The first type chooses the appropriate meaning for a word with multiple meanings, as, for example, the subroutine mentioned above that decides, for names of cities, whether the meaning is Team = A_t or Place = A_p. The second type alters or modifies the attribute or value of an appropriate syntactically related word. For example, one such subroutine puts its value in place of the value of the main noun in its phrase. Thus Team = (blank) in the phrase *each team* becomes Team = each; in the phrase *what team,* it becomes Team = ?. Another modifies the attribute of a main noun. Thus Team = (blank) in the phrase *winning team* becomes $\text{Team}_{(\text{winning})}$ = (blank). In the question "Who beat the Yankees on July 4?", this subroutine, found in the meaning of *beat,* modifies the attribute of the subject and object, so that Team = ? and Team = Yankees are rendered $\text{Team}_{(\text{winning})}$ = ? and $\text{Team}_{(\text{losing})}$ = Yankees. Another subroutine combines these two operations: it both modifies the attribute and changes the value of the main noun. Thus, Game = (blank) in the phrase *six games* becomes $\text{Game}_{(\text{number of})}$ = 6, and in the phrase *how many games* becomes $\text{Game}_{(\text{number of})}$ = ?.

After the subroutines have been executed, the question is scanned to consolidate those attribute-value pairs that must be represented on the specification list as a single entry. For example, in "Who was the winning team . . ." Team = ? and $\text{Team}_{(\text{winning})}$ = (blank) must be collapsed into $\text{Team}_{(\text{winning})}$ = ?. Next, successive scans will create any sublists implied by the syntactic structure of the question. Finally, the composite information for each phrase is entered onto the spec list. Depending on its complexity, each phrase furnishes one or more entries for the list. The resulting spec list is printed in outline form, to provide the questioner with some intermediate feedback.

Processing Routines

PROCESSOR

The specification list indicates to the processor what part of the stored data is relevant for answering the input question. The processor extracts the matching information from the data and produces, for the responder, the answer to the question in the form of a list structure.

The core of the processor is a search routine that attempts to find a match, on each path of a given data structure, for all the attribute-value pairs on the spec list; when a match for the whole spec list is found on a given path, those pairs relevant to the spec list are entered on a *found list.* A particular spec list is considered matched when its attribute has been found on a data path and either the data value is the same as the spec value, or the spec value is ? or *each,* in which case any value of the particular attribute is a match. Matching is not always straightforward. Derived attributes and some modified attributes are functions of a number of attributes on a path and must be computed before the values can be matched. For example, if the spec entry is Home Team = Red Sox, the actual home team for a particular path must be computed from the place and teams on that path before the spec value Red Sox can be matched with the computed data value. Sublists also require special handling because the entries on the sublist must sometimes be considered separately and sometimes as a unit in various permutations.

The found list produced by the search routine is a hierarchical list structure containing one main or derived attribute on each level of each path. Each path on the found list represents the information extracted from one or more paths of the data. For example, for the question "Where did each team play in July?", a single path exists, on the found list, for each team which played in July. On the level below each team, all places in which that team played in July occur on a list that is the value of the attribute Place. Each path on the found list may thus represent a condensation of the information existing on many paths of the search data.

Many input questions contain only one query, as in the question above, *i.e.*, Place = ?. These questions are answered, with no further processing, by the found list produced by one execution of the search routine. Others require simple processing on all occurrences of the queried attribute on the generated found list. The question "In how many places did each team play in July?" requires a count of the places for each team, after the search routine has generated the list of places for each team.

Other questions imply more than one search as well as additional processing. For a spec attribute with the value *every,* a comparison with a list of all possible values for that attribute must be made after the search routine has generated lists of found values for that attribute. Then, since only those found list paths for which all possible values of the attribute exist should remain on the found list as the answer to the question, the search routine, operating on this found list as the data, is again executed. It now generates a new found list containing all the data paths for which all possible values of the attribute were found. Likewise, questions involving a specified number, such as 4 teams, imply a search for *which teams,* a count of the teams found on each path, and a search of the found list for paths containing 4 teams.

In general, a question may contain several implicit or explicit queries. Since these queries must be answered one at a time, several searches, with intermediate processing, are required. The first search operates on the stored data while successive searches operate on the found list generated by the preceding search operation. As an example, consider the question "On how many days in July did eight teams play?" The spec list is

$$\begin{aligned}\text{Day}_{(\text{number of})} &= ?;\\ \text{Month} &= \text{July};\\ \text{Team}_{(\text{number of})} &= 8.\end{aligned}$$

On the first pass, the implicit question *which teams* is answered. The spec list for the first search is

```
Day   = Each;
Month = July;
Team  = ?.
```

The found data is a list of days in July; for each day there is a list of teams that played on that date. Following this search, the processor counts the teams for each day and associates the count with the attribute Team. On the second search, the spec list is

Day = ?;
Month = July;
$\text{Team}_{(\text{number of})}$ = 8.

The found data is a list of days in July on which eight teams played. After this pass, the processor counts the days, adds the count to the found list, and is finished.

RESPONDER

No attempt has yet been made to respond in grammatical English sentences. Instead, the final found list is printed in outline form. For questions requiring a yes/no answer, YES is printed along with the found list. If the search routine found no matching data, NO is printed for yes/no questions, and NO DATA for all other cases.

Discussion

The differences between Baseball and both automatic language translation and information retrieval should now be evident. The linguistic part of the Baseball program has as its main goal the understanding of the meaning of the question as embodied in the canonical specification list. Syntax must be considered and ambiguities resolved in order to represent the meaning adequately. Translation programs have a different goal: transforming the input passage from one natural language to another. Meanings must be considered and ambiguities resolved to the extent that they affect the correctness of the final translation. In general, translation programs are concerned more with syntax and less with meaning than the Baseball program.

Baseball differs from most retrieval systems in the nature of its data. Generally the retrieval problem is to locate relevant documents. Each document has an associated set of index numbers describing its content. The retrieval system must find the appropriate index numbers for each input request and then search for all documents bearing those index numbers. The basic problem in such systems is the assignment of index categories. In Baseball, on the other hand, the attributes of the data are very well specified. There is no confusion about them. However, Baseball's derived attributes and modifiers imply a great deal more data processing than most document retrieval programs. (Baseball does bear a close relation with the ACSI-MATIC system discussed by Miller et al. at the 1960 Western Joint Computer Conference.)

The concept of the spec list can be used to define the class of questions that the Baseball program can answer. It can answer all questions whose spec list consists of attribute-value pairs that the program recognizes. The attributes may be modified or derived, and the values may be definite or queries. Any combination of attribute-value pairs constitutes a specification list. Many will be nonsense, but all can be answered. The number of questions in the class is, of course, infinite, because of the numerical values. But even if all numbers are restricted to two digits, the program can answer millions of meaningful questions.

The present program, despite its restrictions, is a very useful communication device. Any complex question that does not meet the restrictions can always be broken up into several simpler questions. The program usually rejects questions it cannot handle, in which case the questioner may rephrase his question. He can also check the printed spec list to see if the computer is on the right track, in case the linguistic program has erred and failed to detect its own error. Finally, he can often judge whether the answer is reasonable.

Next Steps

No important difficulty is expected in augmenting the program to include logical connectives, negatives, and relation words. The inclusion of multiple-clause questions also seems fairly straightforward, if the questioner will mark off for the computer the boundaries of his clauses. The program can then deal with the subordinate clauses one at a time before it deals with the main clause, using existing routines. On the other hand, if the syntax analysis is required to determine the clause boundaries as well as the phrase structure, a much more sophisticated program would be required.

The problem of recognizing and resolving semantic ambiguities remains largely unsolved. Determining what is meant by the question "Did the Red Sox win most of their games in July?" depends on a much larger context than the immediate question. The computer might answer all meaningful versions of the question (we know of five), or might ask the questioner which meaning he intended. In general, the facility for the computer to query the questioner is likely to be the most powerful improvement. This would allow the computer to increase its vocabulary, to resolve ambiguities, and perhaps even to train the questioner in the use of the program.

Considerable pains were taken to keep the program general. Most of the program will remain unchanged and intact in a new context, such as voting records. The processing program will handle data in any sort of hierarchical form, and is indifferent to the attributes used. The syntax program is based entirely on parts of speech, which can easily be assigned to a new set of words for a new context. On the other hand, some of the subroutines contained in the dictionary meanings are certainly specific to baseball; probably each new context would require certain subroutines specific to it. Also, each context might introduce a number of modifiers and derived attributes that would have to be defined in terms of special subroutines for the processor. Hopefully, all such occasions for change have been isolated in a small area of special subroutines, so that the main routines can be unaltered. However, until we have actually switched contexts, we cannot say definitely that we have been successful in producing a general question-answering program.

INFERENTIAL MEMORY AS THE BASIS OF MACHINES WHICH UNDERSTAND NATURAL LANGUAGE

by Robert K. Lindsay

Participants in the search for intelligent machines frequently disagree on a basic question of strategy in their quest. On the one hand there are those who believe that the major obstacles can be overcome by reliance on the computer's infallible memory, electronic speed, and arithmetic capabilities if these capacities are cleverly employed in sophisticated searching and statistical procedures. On the other hand there are those who feel that the problems of meaning and intuition must be somehow resolved before significant progress will be made, and that these problems are not solely a matter of speed and arithmetic. This issue will only be resolved by demonstration, and yet it is of some importance to decide how to allocate our efforts. This report takes the position that immediate, practical applications can derive from the former approach, but the major problems will be solved only by the latter. To mention a single example, the implementation of information retrieval techniques on present-day computers would be a large step forward, even though the techniques thus far considered have largely been conceptually trivial. The automation of libraries and scientific document files could immediately bring about great savings in valuable human time and effort, plus increased accuracy in literature searching. Kehl (1961) is now implementing a retrieval system which searches for certain combinations of key words in a large corpus and yields references to those documents which contain the proper combinations. Luhn (1958) has used a straightforward statistical procedure to extract key sentences from scientific articles, thus yielding useful abstracts of a sort. The use of these two techniques on a large scale would go a long way toward extracting us from the clutches of the information explosion which is so often discussed.

And yet it is quite clear that these techniques, whose primary advantages derive from using the great speed of the computer, will not produce intelligent machines, or even produce machines which do simple jobs with the intelligence displayed by a human clerk. For even an unintelligent human does more than count frequencies or search for key words. The human displays intelligent features which are generally summed up by saying that he *understands* the *meaning* of what he hears and reads.

The meaning of meaning and the meaning of understanding have never been adequately explicated when applied to human thought processes. How then can we hope to make them precise enough to enable us to build machines which understand meaning? Before attempting to answer this question, let us attempt to sharpen our intuitions by considering more specifically some examples of things which could be done by machines which understand but which would be beyond the capabilities of machines without this ability.

One of the major problems of the many encompassed by artificial intelligence is that of the mechanical translation of natural languages. Many of the early advocates of mechanical translation felt that high-quality translations could be produced by machines supplied with sufficiently detailed syntactic rules, a large dictionary, and sufficient speed to examine the context of ambiguous words for a few words in each direction. No doubt such machines will be able, when the syntactic rules are discovered, to produce fairly good translations, and yet it should be clear that such machines will never produce truly high-quality translations without the aid of pre-editing and postediting by human translators.

Here is one example of a difficulty. We wish to translate "The boy is in the house" and "The boy is in Paris" into French. In the first instance, the preposition "in" is rendered in French as "dans"; in the second sentence, the same preposition is rendered in French as "à." The human translator makes his decision by knowing that houses enclose people on all sides, while cities do not. This situation could, of course, be handled by marking all nouns with an indicator which tells the machine whether or not the thing denoted can enclose other things. But the hope that all such idiosyncrasies can be handled by such multiplication of stored details is futile. Bar-Hillel (1960) has given an even more perplexing example. We wish to translate the sentences "The pen is in the box" and "The box is in the pen" into French. There is no other context, and no other is needed by a human translator who knows that "pen" in the first sentence must denote a writing instrument and not a fenced enclosure, while the opposite is true in the second sentence. He can thus select the proper French equivalent for each, even though in French a single word does not suffice as it does in English. Once again we must increase the information stored in our machine, this time indicating for each noun those things which it can enclose.

The problem of understanding may be rephrased to state that we must find ways of storing large amounts of such detailed knowledge while keeping the amount of memory capacity required within realizable limits.

Much of the literature on meaning has not been directly connected with this notion, but has been concerned with the problem of denotation: to what things does a symbol refer. Here, however, we are faced with the problem of what a *proposition* means. Osgood (1957) and Mowrer (1954) have attempted to extend notions of association and conditioning to include associations between groups of words rather than single words.

According to Osgood's theory, a word elicits associated internal responses. These responses can be described by their values along certain dimensions, such as active-inactive, good-bad, strong-weak. The meaning of a combination of words is an average of the component values for each of the words taken individually. For example, if "shy" is valued as mildly inactive, mildly bad, and mildly weak, and if "secretary" is valued as being very active, very good, and mildly weak, then "shy secretary" is valued as mildly active, mildly good, and mildly weak.

According to Mowrer's theory, sentences are temporal sequences of words and the internal responses to the first words are conditioned, in the classical sense, to the internal responses to the later words. Thus the sentence "Tom is a thief" conditions the notion of Tom to the notion of thief, where we use the loose term "notion" to indicate that it is not the words themselves, but the internal responses to them which become associated.

In both of these theories, the measure of meaning of a concatenation of words amounts to some sort of combination of the measures for the individual words. It appears that a useful theory must somehow make use of more complicated associative connections than those proposed by either of these two workers. For one thing, Osgood's scheme depends not at all on word order, only on which words are used; Mowrer's scheme depends only on word order, and not upon any other relations. To Osgood, "Tom hit Joe" would have the same meaning as "Joe hit Tom"; to Mowrer, "Tom is a thief" would have the same meaning as "Tom hit a thief."

Intuitively, a concept of meaning must include the notion of implication: what does a proposition imply. This does not mean, that is, imply, logical implication, but merely implication to the individual. Thus the meaning of a proposition is relative to the audience, and this probably is an unavoidable requirement.

Knowing more than one is told is a characteristic of human performance which is present in most behaviors which are called intelligent. We have argued that this characteristic is necessary for machines which are to solve the real problems of information retrieval, language translation, and problem-solving. And furthermore, we must find efficient ways to store implications if we are to develop intelligent machines with finite memory capacities, that is, if we are to develop intelligent machines.

Examples of memory structures with these desired properties immediately come to mind from personal experience. They are often called mental pictures. Gelernter (1959*b*), for example, has developed a geometry machine whose basic source of intelligence lies in its ability to reject most of the formally possible sequences of proof steps because they "cannot possibly be correct." In effect, the machine constructs a diagram based upon the premises of the theorem to be proved. (Actually, the machine is supplied with such a diagram, although the task of constructing one is, while difficult, not taxing of memory and speed.) The implications of the premises are explicitly contained in this diagram, as are some nonimplications, but most nonimplications are not contained. The machine then merely rejects as possible subgoals (intermediate steps) all things which are not true in the diagram. For example, the premises "Triangle ABC," "AB = BC," "A, D, C collinear," and "AD = DC" are supplied in conjunction with coordinates for each point, such as A(0,0), B(5,5), C(10,0) and D(5,0) (see Fig. 1). The machine will never attempt to prove "triangle ABC congruent to triangle ABD" because this is not true in the diagram, as the machine can determine by calculating their respective perimeters. However, it might try to prove "angle DAB = angle DCB," which is implied by the premises. It may also try to prove "AD = DB" which is true in the diagram, but not implied by the premises. By supplying the machine with a more general diagram, such as by moving B to (5,15) (see Fig. 2), this last *cul de sac* could be avoided.

Such two-dimensional pictures have the properties we desire: they store implications and they do so in compact fashion. They also have a wide

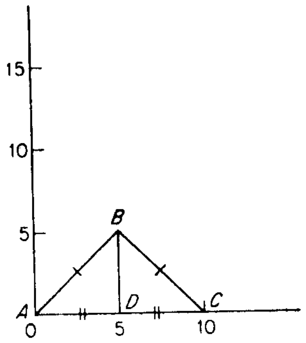


Figure 1.

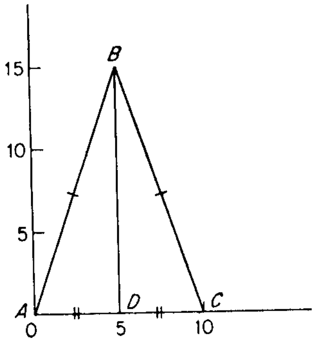


Figure 2.

range of application, few of which, aside from Gelernter's work, have been explored. One further example is provided by Venn diagrams, which are devices which aid logical reasoning. We may represent the proposition "all B are A" by two areas, one completely enclosed by the other, with appropriate labels. If we add a third area, C, according to the same rules to represent the proposition, "all C are B," the resulting diagram contains the implication that "all C are A," since area C must lie wholly within area A. Similar rules can be devised for other propositional forms, as elsewhere discussed by Lindsay (1961). Actually there are simpler schemes for such situations since not all of the properties of euclidian two-dimensional space are required, but, since such representations also handle many other situations, an intelligent machine would achieve some economy by employing such general-purpose representations wherever usable rather than devising special schemes for each case.

So far we have discussed only situations where few difficulties arise. Reasoning does not always obey the rules of logic and geometry, and we quickly encounter additional difficulties when we attempt to handle even simple situations. A program has been written to handle a different class of problems, and the difficulties will become clear as this program is described.

The program to be described parses sentences written in Basic English and makes inferences about kinship relations. To do this it constructs two types of complex structures in the computer memory, one corresponding to a sentence diagram of the sort produced by high-school students, the other corresponding to the familiar family tree. These are represented inside the computer by so-called *list structures*. A list structure is a form of associative memory, wherein each symbol is tagged by an indicator which tells the machine the location of a related symbol. So far this corresponds to the associative bonds which are the basic concept of stimulus-response psychology. However, each symbol may at times refer to a whole string of other, connected symbols, thus producing a hierarchical organization of memory associations. This feature provides much greater flexibility than either the single associations of stimulus-response psychology or the mediated associations which have recently been discussed and seem to be a first step in the direction of generalizing the limited stimulus-response schema.

The Sentence-parsing Program

The grammars of certain languages may be described by rules of replacement, which, if they satisfy certain conditions of simplicity, are called phrase structure rules (see Chomsky, 1957). For example, a simple grammar might consist of the following rules:

1. S ↔ NP + Pred
2. S ↔ NP + VP + NP
3. S ↔ NP + V + NP
4. NP ↔ They
5. NP ↔ planes
6. NP ↔ A + N
7. N ↔ planes
8. N ↔ man
9. A ↔ the
10. A ↔ flying
11. VP ↔ Aux + V
12. V ↔ are
13. V ↔ flying
14. Aux ↔ are
15. Pred ↔ VP + NP + Ad
16. Ad ↔ swiftly

These rules may be interpreted as, for example, the twelfth, "When V is encountered in a string of symbols, it may be replaced by 'are,'" or "when 'are' is encountered in a string of words it may be replaced by V." The former interpretation concerns the production of sentences, while the latter concerns the parsing of sentences. Thus we may produce a sentence by beginning with the symbol S and successively applying rules. For example: S → NP + VP + NP → NP + Aux + V +NP → They + Aux + V + NP → They + are + V + NP → They + are + flying + NP → They are flying planes.

Different sequences of rules produce different sentences. With the rules given above, certain sentences can be produced which are ungrammatical within English. For example, we could generate "The man are flying planes." A proper grammar (set of rules) for English would have to rule out such possibilities. This is generally accomplished both by defining rules more narrowly (assuring, for example, that subject agrees with verb) and by introducing certain metarules which specify which sequences of application are legitimate [for two methods of accomplishing this, see Chomsky (1957) and Pendergraft (1961)].

It is also clear that different sequences of rules may produce the same sentences. For example:[1] S → NP + V + NP → NP + V + A + N → They + V + A + N → They + are + A + N → They + are + flying + N → They are flying planes.

Consider now a straightforward parsing technique which might be applied to the sentence "They are flying planes swiftly," using the rules of our example. This sentence has a unique parsing which may be discovered as follows. Find each word or group of words which occurs in the sentence

[1] This example is due to Chomsky (1957).

on the right-hand side of one of the rules, and replace it by the symbol which appears on the left-hand side of the rule. Apply the same procedure to the resulting string of symbols. If a symbol or word appears on the right-hand side of more than one rule, form separate strings for each case. Continue until the sequence is reduced to the single symbol S, abandoning paths to which this procedure ceases to apply. Thus we have:

```
         They are flying planes swiftly
1.       NP + V + V + N + Ad
           can go no farther
2.       NP + V + V + NP + Ad
           can go no farther
3.       NP + V + A + N + Ad
         NP + V + NP + Ad
         S + Ad
           can go no farther
4.       NP + V + A + NP + Ad
           can go no farther
5.       NP + Aux + A + N + Ad
         NP + Aux + NP + Ad
           can go no farther
6.       NP + Aux + A + NP + Ad
           can go no farther
7.       NP + Aux + V + N + Ad
         NP + VP + N + Ad
           can go no farther
8.       NP + Aux + V + NP + Ad
         NP + VP + NP + Ad
   8.1.      S + Ad
               can go no farther
   8.2.      NP + Pred
             S
               successful parsing
```

Even after a sufficient number of rules and metarules are supplied so as to eliminate ungrammatical sequences, there will remain, for natural languages, sentences which can be generated by two or more different production sequences. Conversely, any procedure which parses sentences should be able to discover all such sequences. The decision as to which parsing is correct depends upon a context larger than a single sentence and in many cases will also depend upon the meaning of the sentence, including a dependence upon what the various words denote.

However, even if we neglect the problem of selecting which legitimate parsing is correct in a given instance, the problem of discovering *any* legitimate parsing is itself formidable when we deal with the tens of thousands of rules needed to describe a natural language. A complete set of rules is, in fact, so large that none has yet been devised for any natural language, although some have been under study for thousands of years. With even a moderately large number of rules, the parsing procedure described above will generate many possible branches, some of which may continue to be feasible for a long time. In order to discover any parsing in reasonable time and with reasonable effort, it is useful to employ some sort of selection procedure.

The procedure employed in the program to be described here is based upon two assumptions which are psychologically realistic. First, it is assumed that almost all sentences which will be encountered in actual text may be parsed by a procedure which proceeds from left to right, making decisions about the disposition of earlier phrases without considering the entire sentence. This reduces the number of rule combinations which must be searched. Secondly, it is assumed that a very limited amount of memory is available to remember intermediate results during the parsing of even extremely long sentences. This places severe restrictions upon which types of complexity will be analyzed and which types of syntactic structure will not be handled.

The final result of applying the parsing program to a sentence is an associatively organized memory whose structure reflects the interrelations among words, but does not give complete information as to which rules should be applied first to produce the sentence. The sentence diagram for our above example might be drawn as in Fig. 3. Each node in a sentence diagram corresponds to a substructure which has been constructed during the scanning of the sentence.

The sentence-parsing program is provided with a string of words as an input sentence. Each word may be found in a dictionary which indicates

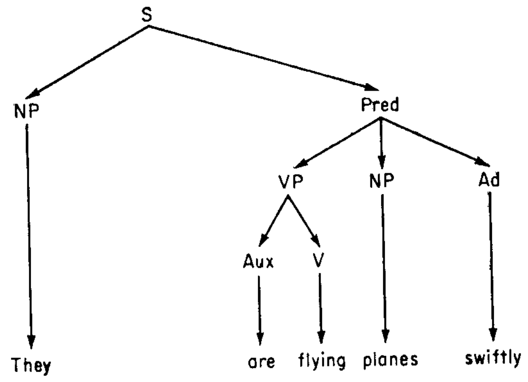


Figure 3.

a series of possible parts of speech which that word may serve as, the series being ordered so as to present the most frequent function first. The machine proceeds in a left-to-right fashion, assuming temporarily that the first word serves its most common function. To each part of speech there corresponds an associative structure which the machine forms. This structure is then temporarily held in memory and the next word is examined. If the structure so created requires the services of an additional word type, the machine continues to search for this type. If the next word serves the purpose, its structure is incorporated with that of the first word so that only a single structure name must be kept in rapid access memory, the remainder of the information being obtainable via the name. If the next word does not serve the desired purpose, its structure is stored separately, and the machine continues to look for words which will complete the structures of each of the words now held in memory. However, the number of structure names which can be held for rapid access is limited to a small total so that the machine must eventually begin to combine its substructures or else forget where it is. Frequently, the machine will be forced to complete a structure even though it has not found what it wants. This results in changing the part of speech designation for one or more words so that the entire sequence will now be compatible. The machine thus proceeds through the sentence, making temporary decisions, storing substructures in its limited rapid access memory, and revising its decisions only when forced to by lack of rapid access memory space or by complete incompatibility of substructures.

Loosely, the machine's behavior can be described as follows. The first word is "the." All right, now I need to find a nounlike word. The second word is "very," so now I need an adjective or adverb. The third word is "big," which is the adjective I needed, so combine these two words into the structure "very big." Now I need a nounlike word. The fourth word is "man," which is the noun needed. Now all words are combined into the structure "(the((very) big)) man." But now we have a subject, so look for a verb. The fifth word is "bit" which can act as the verb, so create the structure "((the((very)big))man) bit." Now another nounlike word or structure could serve as object. The sixth word is "the," so save it to modify a nounlike word; now we have two things saved, both looking for a nounlike word. The final word is "dog" which will serve both needed functions. We now have the complete sentence, whose structure is illustrated in Fig. 4.

In one sense this program is nothing but an algorithm which produces an output for every possible sequence of part-of-speech series inputs. However, if the program were written so as to merely check the input sequence and produce the corresponding output, a serious difficulty would arise. The size of the table required increases exponentially with sentence length,

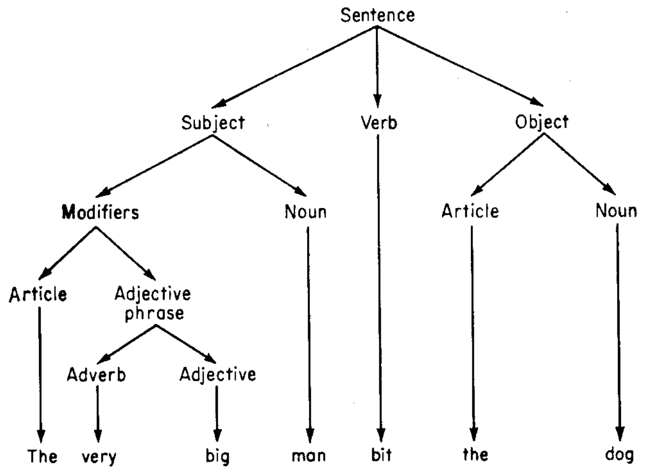


Figure 4.

thus an extremely large table for sentences of even moderate length would be needed. Although the program as it stands is limited in terms of the complexity of the sentences which it can handle, length alone does not contribute to complexity. For example, the program can handle a sentence such as "The big, black, ugly, ferocious, . . . , strong dog bit him," where the number of adjectives which may be inserted in the string is limited only by the memory capacity of the computer. This is possible because all of the adjectives are combined into a single substructure at every step, hence the rapid access memory is never exceeded. Further, the total memory requirements only increase linearly with sentence length. However, sentences which require the construction of an excessive number of substructures will cause the program to fail, even though these sentences are relatively short. Yngve (1961) has described a similar device for *producing* sentences, and argues that the limitations on complexity imposed by limits on rapid access memory capacity explain why certain constructions are not commonly used in natural English and hence are called ungrammatical.

The part-of-speech routines provide a finite set of processes which can handle an infinite number of sentences, in principle. They are superior to the table look-up method for the same reason that a program which computes e^x for any value of x is superior to a table of this function and a look-up program.

Obviously humans must employ some finite set of processes which are used to parse sentences, and obviously each word acts as a stimulus to elicit the corresponding processes. These processes, as in this program, are highly complex, and their decisions are contingent upon which other processes have been initiated before. It is quite consistent with our knowledge of human thought processes to assume that the interaction takes place in the above-described manner, that is, through a small set of rapid access memory locations. However, the adult human undoubtedly has a larger set of processes, which effectively categorize words into more narrow categories than the few part-of-speech designations provided to our program, and these processes are no doubt much more complex.

Another psychologically tenable feature of this program is its left-to-right analysis. Although English grammar may conceivably be more readily analyzed in some other fashion, humans generally proceed from left to right, with only occasional reversals.

The limits of this program are not very well known. It will accept no words not included in Basic English, a system of grammar and a vocabulary of about 1700 English words which was defined by Ogden (1933). (Basic English is simplified English in the sense that anything which is good Basic is good English, but not vice versa.) The program will not accept certain punctuation marks, such as colons, and it does not distinguish between others, such as exclamation points and periods. Also, phrases and clauses in apposition must be indicated by dashes rather than commas. However, the program is not limited to single-clause sentences, nor must the input be a complete sentence. Thus, the program can handle many inputs which appear in actual writing but not in books on grammar. The program always makes a decision, and the result is always a complete structure containing all of the input.

The end result of the application of the program to a sentence is a structure which relates all the words of a sentence. This could be replaced logically by a set of descriptions listing all of these relations, but such a set would be far more elaborate and costly to memory. The syntactic meaning of the sentence is just this structure, wherein relations among words are implicit in its organization.

The Semantic-analysis Program

After the diagram of a sentence is constructed, the program attempts to deal with the meaning of some of the words. First, a list of all nouns is constructed. This list includes not only words which were used as subjects or objects, but also names used as possessive adjectives, such as "Bill's."

At this point, words cease to be considered solely by their syntactic-category membership. The sentence diagram is used as an information store which relates words. Subject-object combinations whose main verb is some form of "to be" are discovered. When such a combination is found, the words are marked "equivalent" by a cross-referencing scheme which indicates that both subject and object refer to the same thing or person. The modifiers of all equivalent nouns are then grouped together.

Next, a search is made for the eight words which Basic English provides to discuss kinship relations: "father," "mother," "brother," "sister," "offspring," "brother-in-law," "sister-in-law," and "married." If any of these relation words occurred in the sentence, their modifiers are examined to discover proper names which appeared as possessive adjectives or objects of a preposition, as would be the case if the original sentence contained phrases of the form "Jane's brother" or "the father of John." Each such proper name is paired with all others associated with the same occurrence of the relation word. By proceeding through the entire collection of words in this fashion, a list of elementary relations is formed. The items on this list are word triplets, two proper nouns, and a relation word which connects them.

Now the family tree is constructed. The computer memory is organized in an associative fashion again, with one computer storage location linked to others. The structure is isomorphic to diagrams such as given in the example below. Each "marriage" is represented by an association between the husband and wife, plus the name of a similar family unit for the parents of the husband, another for the parents of the wife, and the name of a list of offspring of this marriage. If the names of one of the partners, one of their parents, or some of their offspring are not given, places are reserved for these names should they occur in the future. The resulting tree is the same no matter whether the information was explicitly given in the text or merely implied.

By way of illustration, Fig. 5 depicts the memory structure for a simple family tree. The tree is composed of two basic family units, one formed by the marriage of A and B, and the other formed by the marriage of C and D. One of the offspring, E, of the first marriage is married to one of the offspring, F, of the second. It is evident that many relations are described by the tree given. However, it is important to note that the associations are one-way associations. This fact necessitates the addition of the name of the parent family unit at the end of each offspring unit. Thus, given A we may determine that E is one of his offspring by moving only in the direction of the arrows. Given E we may again trace the connection to A by moving only in the direction of the arrows, but this is true only because the family unit associated with E also contains the name of A's family unit. It follows that, given the fact that F (already located in the tree) and G (a name not previously encountered) are siblings, it is not sufficient to add G to the list of A-B offspring (dotted lines) but we must also copy the name of F's parental family unit into the newly constructed family unit of G (dashed lines).

The family tree, or trees, so constructed, are not erased after a single sentence is processed, but continue to grow as additional information is given throughout the passage.

The complexities and many small difficulties which are encountered in even this simple type of relation are indicative of the problem involved in the construction of semantic structures. More instructive, however, are the conceptual problems which arise in attempting to generalize this program to less strictly structured situations. Let us consider two of the most important problems.

It often happens, even when dealing with simple kinship relations, that the order of presentation of the input information has a crucial effect upon the efficiency of memory allocation. For example, if we are first told that X has offspring A, B, C, and D, we must construct an elaborate organization to handle this information, locations such as for the spouse of X being left blank. If we are then told that Y has offspring E, F, G, and H, we must construct another such structure, unrelated to the first. Finally, we may learn that B and H are brothers. This permits (neglecting such complications as multiple marriages) a collapsing of the two structures into a single organization which much more compactly represents the information

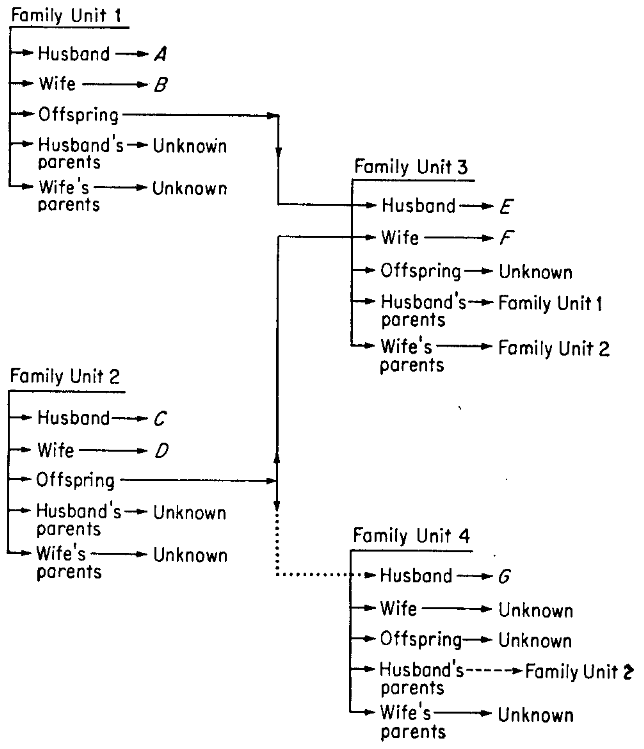


Figure 5.

implied. If we had been fortunate enough to have first learned of B's relation to H (or of X's relation to Y), we would have made much more efficient use of our memory capacity. In the program, the extra structures are "erased," that is, the memory used for them is returned to a common stockpile for use anywhere else it is needed. This is quite handily done with the easily altered computer memory, but a memory which is hard to erase, as the human memory presumably is, could be affected in important ways by such unhappy input sequences.

Nonetheless, an intelligent machine should have the property of being intelligent no matter what the order of its inputs. One aspect of the "aha" phenomenon is just that many formerly unrelated items of information are suddenly brought together by a single additional item, so that many implications suddenly leap out. Educators are beset by the problem of determining optimal orders of presentation of material, but, fortunately for the student, the human mind is capable of seeing connections under nonoptimal conditions.

An even more baffling problem is that of handling what has been called connotative meaning (Lindsay, 1961). Probably more often than not, a set of propositions which make some definite implications contains several subsets which alter the probabilities of other propositions without making any of them definite. Thus the statement that "George voted for Eisenhower and is opposed to medical care for the aged" makes it more likely that George is opposed to the United Nations, though only slightly so. It is quite clear that human cognitive organizations frequently take cognizance of these altered probabilities, perhaps to a greater extent than is reasonable. But how can such implicit connotations be intelligently and efficiently handled?

Let us consider a more concrete example arising in the context of the kinship-relation program. Consider the following sentence: "Joey was playing with his brother Bobby in their Aunt Jane's yard when their mother called them home." Certain definite information is given by this sentence, such as the fact that Joey and Bobby are brothers. Also, it is clear that Jane is either the sister or the sister-in-law of the children's mother, but it is not known definitely which is the case. If previous information has related, say, Joey to many others and Jane to many others, but has not related Joey's relations to Jane's relations, then the given sentence may imply a large number of things and remove the possibilities of a large number of other things. Still other possibilities depend upon knowing the exact relation between Joey's mother and Jane. The problem is to capture in the family-tree structure all of the definite implications, to eliminate all of the things definitely ruled out, indicate the altered probabilities of other relations, and still not make any *definite* assertions about the relation between Joey's mother and Jane.

The structure thus far described is unable to handle even this simple case, since the associations are either there or they are not, and only one connection is permissible. One solution to this problem is to construct several family-tree structures, one for each possibility. This corresponds, for example, to the situation in which a student will draw diagrams of an acute triangle, a right triangle, and an obtuse triangle corresponding to the possible cases for which he wishes to prove a theorem. This solution, however, will work well only when the number of alternatives is small and when the structures are themselves simple. In more complex situations this procedure is too taxing of memory capacity. It is desirable to include the uncertainty within a single structure.

In order to do this, we must allow multiple connections. Thus, in place of every association in our original format we must substitute a list of all the possibilities, and the process which retrieves information must recognize that only one of these can be correct. When nothing at all has been implied, the lists of possibilities will contain an "all" symbol indicating that all things are possible; when something definite is implied later, this "all" symbol is replaced by the proper connection; when several things have been implied, the universal symbol is replaced by a list of the remaining possibilities. We may also need to record a list of connections which are definitely impossible. When nothing has been implied, this list will contain a "none" symbol indicating that no things are impossible.

It is to be noted that a probabilistic connection of the sort frequently hypothesized by psychologists is not appropriate here. That is, we do not want a connection such that a given stimulus will sometimes evoke one association, sometimes another on a probability basis. In the above example, the reader knows *definitely* that either Jane is the sister of Joey's mother *or* is the sister of Joey's father, but not both; no reader would conclude half the time that Jane is the sister of Joey's mother and half the time that she is the sister of Joey's father, altering his decision from time to time.

But we are still faced with two problems. First, it is impossible, or at least impractical, to retain an extremely large number of possibilities; second, it is not clear how we should indicate that some possibilities are more probable than others. The first problem is perhaps solved by humans by not remembering all possibilities; thus humans are unable to remember all possible implications when the set of such possibilities is large. This will no doubt remain a problem for machines as well. We might solve the second problem by ordering the list of probabilities, placing the most likely alternatives first; or perhaps we might decide to associate probabilities with each alternative. In any event, the probabilities so established will determine the weight which is assigned to implications, but will *not* determine the implications which will hold to the exclusion of others. Finally, we can imagine a situation in which the list of possibilities is truncated due to the limited computing capabilities of man or machine, and where subsequently all of the possibilities which remain are eliminated by further information. In this case, the machine, after all, will have to indicate that something is wrong and review previous inputs, this time reselecting possibilities in the light of knowledge of information to follow.

To complete our example, we may present the modified storage format (Fig. 6) which could be used to solve our sample problem.

Finally, we note that we have solved the problem of connotative meaning while retaining our basic device of storing *definite* implications implicitly, but we have resorted to storing *possible* implications explicitly. Techniques for avoiding this listing of possibilities would prove extremely valuable, since as we have seen, requirements on memory capacity increase rapidly when storage is explicit.

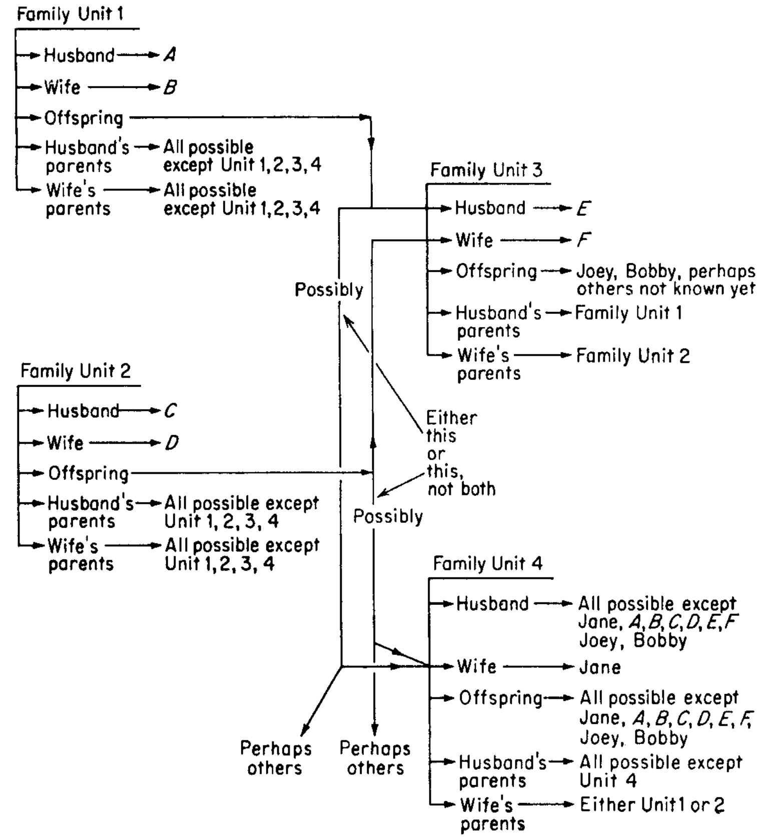

Family Unit 1
Husband → A
Wife → B
Offspring
Husband's parents → All possible except Unit 1,2,3,4
Wife's parents → All possible except Unit 1,2,3,4
Family Unit 3
Husband → E
Wife → F
Offspring → Joey, Bobby, perhaps others not known yet
Husband's parents → Family Unit 1
Wife's parents → Family Unit 2
Possibly
Either this or this, not both
Possibly
Family Unit 2
Husband → C
Wife → D
Offspring
Husband's parents → All possible except Unit 1, 2, 3, 4
Wife's parents → All possible except Unit 1, 2, 3, 4
Family Unit 4
Husband → All possible except Jane, A, B, C, D, E, F Joey, Bobby
Wife → Jane
Offspring → All possible except Jane, A, B, C, D, E, F, Joey, Bobby
Husband's parents → All possible except Unit 4
Wife's parents → Either Unit 1 or 2
Perhaps others
Perhaps others

Figure 6.

Summary

It has been argued that the problem of meaning is of major importance in the study of the nature of intelligence, and that a useful definition of meaning must include not only denotation but connotation and implication as well. To handle these important questions it is necessary to study cognitive organizations which are more complex than those upon which most psychological theories are based. A central question is the storage of large numbers of interrelated propositions in a manner which efficiently uses memory capacity. Illustration of these points was given by reference to a computer program which stores syntactic relations and extracts and stores semantic implications of a very limited character. The illustrations put into concrete terms some of the problems which must be resolved before machines of formidable intelligence can be constructed.

section 6

Pattern Recognition

Pattern-recognition research is one of the most difficult areas of artificial intelligence activity to characterize succintly. In its infancy it was concerned with optical character recognition and later, voice recognition. Selfridge has labeled this kind of pattern-recognition work as "eyes and ears for computers." More recently the "pattern-recognition" label has been applied, with much justification, to studies of hypothesis formation by machine, discrimination learning in random nets, perceptual learning in human beings and schemes for inductive inference automata in general. There is a close relationship between the work on pattern recognition and studies of cognitive behavior. It is quite appropriate, therefore, that this material should be placed at the junction of the two major sections of this volume.

Much pattern recognition research has been concerned with programming a computer to recognize that the same name should be given to different manifestations of an object, for example, that short A's, tall A's, fat A's, and skinny A's are all A's. This behavior might be described as elementary perceptual generalization.

The problem becomes one of being able to represent the essence of A-ness in a computer program. For some researchers the computer is a vehicle for testing out their hypotheses about A-ness (perceptual performance). The second example in the Selfridge-Neisser report is a case in point. For others, the computer is a vehicle for generating hypotheses about A-ness (perceptual learning). The Uhr-Vossler piece is an example of this approach.

The work in pattern recognition differs from the work in computer models of verbal learning, hypothesis behavior, and particularly concept formation only in the relatively greater emphasis on complex "central" cognitive processes in the latter work. However the importance of the basic pattern-recognition activity in problem-solving by human beings or computers is well recognized. The principal function of pattern recognition may be characterized as reduction of complex environments. Neither the human being nor the computer can afford to deal with each event as a special case. Suppose, for example, that a performance system has stored some useful "A information." This information about A may be independent of the size or shape of A. If tall A, short A, skinny A, and fat A can all be recognized as A, the stored information can be made available about any particular occurrence of an A. A small set of information structures and processes can be made powerful if there is available a device for recognizing when and where this set is applicable or relevant.

Even if we classify the reports of Feigenbaum, Hunt and Hovland, and Feldman as reports on pattern recognition, we still have collected in this volume only a very small sample of the available work on the subject. This inadequacy is remedied somewhat by the short review in the Uhr-Vossler study and by the discussion in Minsky's review. We regret that space limitations forced the omission of the pioneering work of Selfridge (1955). His report was an early landmark in pattern recognition in particular and artificial intelligence in general. The work of Kochen (1961*a*, 1961*b*) is also of great interest and closely related to work in concept formation. We have also omitted reports on pattern recognition in randomly connected nets. We hope that the Bibliography will provide some help to the reader who wants to investigate these and others works on pattern recognition.

Oliver Selfridge is on the staff of the Lincoln Laboratory, Massachusetts Institute of Technology, and Ulric Neisser is a member of the faculty of the Psychology Department, Brandeis University.

Leonard Uhr is on the staff of the Mental Health Research Institute, University of Michigan, and Charles Vossler was a member of the Artificial Intelligence Research Staff of the System Development Corporation, Santa Monica, when this research was done. Vossler is now at Cornell Aeronautical Laboratory, Buffalo, New York.

PATTERN RECOGNITION BY MACHINE

by Oliver G. Selfridge & Ulric Neisser

Can a machine think? The answer to this old chestnut is certainly "yes": Computers have been made to play chess and checkers, to prove theorems, to solve intricate problems of strategy. Yet the intelligence implied by such activities has an elusive, unnatural quality. It is not based on any orderly development of cognitive skills. In particular, the machines are not well equipped to select from their environment the things, or the relations, they are going to think about.

In this they are sharply distinguished from intelligent living organisms. Every child learns to analyze speech into meaningful patterns long before he can prove any propositions. Computers can find proofs, but they cannot understand the simplest spoken instructions. Even the earliest computers could do arithmetic superbly, but only very recently have they begun to read the written digits that a child recognizes before he learns to add them. Understanding speech and reading print are examples of a basic intellectual skill that can variously be called cognition, abstraction or perception; perhaps the best general term for it is pattern recognition.

Except for their inability to recognize patterns, machines (or, more accurately, the programs that tell machines what to do) have now met most of the classic criteria of intelligence that skeptics have proposed. They *can* outperform their designers: The checker-playing program devised by Arthur L. Samuel of International Business Machines Corporation (1959*a*) usually beats him. They *are* original: The "Logic Theorist," a creation of a group from the Carnegie Institute of Technology and the RAND Corporation [Newell, Simon, and Shaw (1956*a*, 1957*b*)] has found proofs for many of the theorems in *Principia Mathematica,* the monumental work in mathematical logic by A. N. Whitehead and Bertrand Russell (1940). At least one proof is more elegant than the Whitehead-Russell version.

Sensible as they are, the machines are not perceptive. The information they receive must be fed to them one "bit" (a contraction of "binary digit," denoting a unit of information) at a time, up to perhaps millions of bits. Computers do not organize or classify the material in any very subtle or generally applicable way. They perform only highly specialized operations on carefully prepared inputs.

In contrast, a man is continuously exposed to a welter of data from his senses, and abstracts from it the patterns relevant to his activity at the moment. His ability to solve problems, prove theorems and generally run his life depends on this type of perception. We suspect that until programs to perceive patterns can be developed, achievements in mechanical problem-solving will remain isolated technical triumphs.

Developing pattern-recognition programs has proved rather difficult. One reason for the difficulty lies in the nature of the task. A man who abstracts a pattern from a complex of stimuli has essentially classified the possible inputs. But very often the basis of classification is unknown, even to himself; it is too complex to be specified explicitly. Asked to define a pattern, the man does so by example; as a logician might say, ostensively. This letter is A, that person is mother, these speech sounds are a request to pass the salt. The important patterns are defined by experience. Every human being acquires his pattern classes by adapting to a social or environmental consensus—in short, by learning.

In company with workers at various institutions our group at the Lincoln Laboratory of the Massachusetts Institute of Technology has been working on mechanical recognition of patterns. Thus far only a few simple cases have been tackled. We shall discuss two examples. The first one is MAUDE (for Morse Automatic Decoder), a program for translating, or rather transliterating, hand-sent Morse code. This program was developed at the Lincoln Laboratory by a group of workers under the direction of Bernard Gold.

If telegraphers sent ideal Morse, recognition would be easy. The keyings, or "marks," for dashes would be exactly three times as long as the marks for dots; spaces separating the marks within a letter or other character (mark spaces) would be as long as dots; spaces between characters (character spaces), three times as long; spaces separating words (word spaces), seven times as long. Unfortunately human operators do not transmit these ideal intervals. A machine that processed a signal on the assumption that they do would perform very poorly indeed. In an actual message the distinction between dots and dashes is far from clear. There is a great deal of variation among the dots and dashes, and also among the three kinds of space. In fact, when a long message sent by a single operator is analyzed, it frequently turns out that some dots are longer than some dashes, and that some mark spaces are longer than some character spaces. (See Fig. 1.)

With a little practice in receiving code, the average person has no trouble with these irregularities. The patterns of the letters are defined for him in terms of the continuing consensus of experience, and he adapts to them as he listens. Soon he does not hear dots and dashes at all, but perceives the characters as wholes. Exactly how he does so is still obscure, and the mechanism probably varies widely from one operator to another. In any event transliteration is impossible if each mark and space is considered individually. MAUDE therefore uses contextual information, but far less than is available to a trained operator. The machine program knows all the standard Morse characters and a few compound ones, but no syllables or words. A trained operator, on the other hand, hears the characters themselves embedded in a meaningful context.

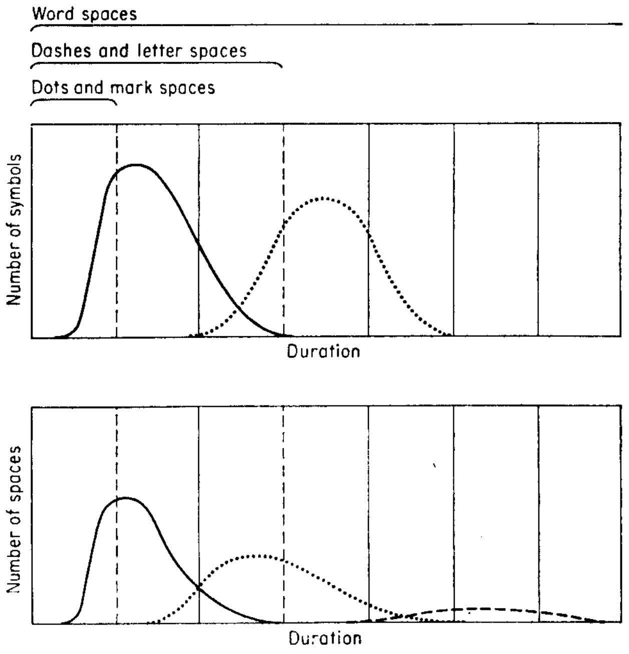


Figure 1. Variability of Morse code sent by a human operator is illustrated in these curves. Upper graph shows range of durations for dots (solid curve) and dashes (dotted curve) in a message. Lower graph gives the same information for spaces between marks within a character (solid curve), spaces between characters (dotted curve) and between words (dashed curve). Ideal durations are shown by brackets at top and vertical broken lines.

Empirically it is easier to distinguish between the two kinds of mark than among the three kinds of space. The main problem for any mechanical Morse translator is to segment the message into its characters by identifying the character spaces. MAUDE begins by assuming that the longest of each six consecutive spaces is a character space (since no Morse character is more than six marks long), and the shortest is a mark space. It is important to note that although the former rule follows logically from the structure of the ideal code, and that the latter seems quite plausible, their effectiveness can be demonstrated only by experiment. In fact the rules fail less than once in 10,000 times.

The decoding process is as follows. (See Fig. 2.) The marks and spaces, received by the machine in the form of electrical pulses, are converted into a sequence of numbers measuring their duration. (For technical reasons these numbers are then converted into their logarithms.) The sequence of durations representing spaces is processed first. The machine examines each group of six (spaces one through six, two through seven, three through eight and so on), recording in each the longest and shortest durations. When this process is complete, about 75 per cent of the character spaces and about 50 per cent of the mark spaces will have been identified.

To classify the remaining spaces a threshold is computed. It is set at the most plausible dividing line between the range of durations in which mark spaces have been found and the range of the identified character spaces. Every unclassified number larger than the threshold is then identified as a character space; every one smaller than the threshold, as a mark space.

Now, by a similar process, the numbers representing marks are identified as dots and dashes. Combining the classified marks and spaces gives a string of tentative segments, separated by character spaces. These are inspected and compared to a set of proper Morse characters stored in the machine. (There are about 50 of these, out of the total of 127 possible sequences of six or fewer marks.) Experience has shown that when one of the tentative segments is not acceptable, it is most likely that one of the supposed mark spaces within the segment should be a character space instead. The program reclassifies the longest space in the segment as a character space and examines the two new characters thus formed. The procedure continues until every segment is an acceptable character, whereupon the message is printed out.

In the course of transmitting a long message, operators usually change speed from time to time. MAUDE adapts to these changes. The computed thresholds are local, moving averages that shift with the general lengthening or shortening of marks and spaces. Thus a mark of a certain duration could be classified as a dot in one part of the message and a dash in another.

MAUDE's error rate is only slightly higher than that of a skilled human

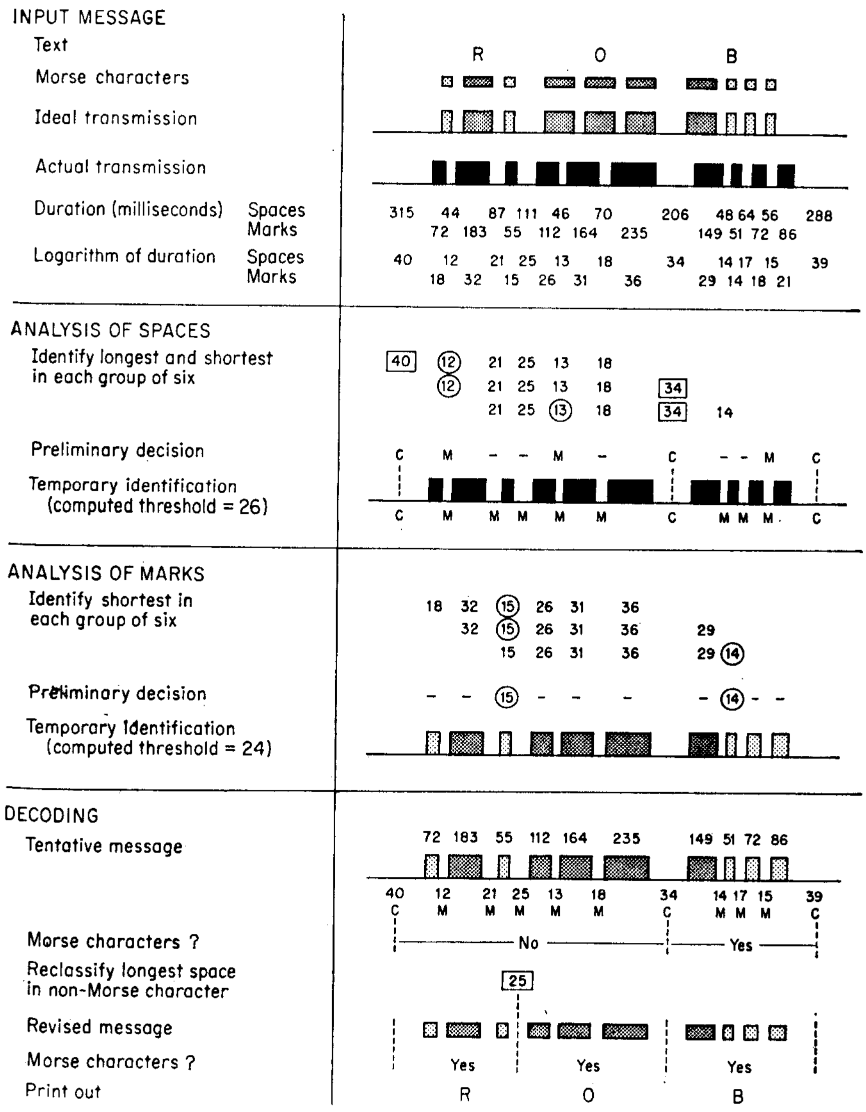


Figure 2. "MAUDE" program described in text, translates Morse code. Marks identified as dots are shown in light color; marks identified as dashes in dark color. Unidentified marks are in black. Character spaces are denoted by C; mark spaces, by M. A circle around a number indicates that it is the smallest in a group; a rectangle means it is the largest. Analysis of spaces and marks proceeds by an examination of successive groups of six throughout the message. The table shows only the first three such groups in each case.

operator. Thus it is at least possible for a machine to recognize patterns even where the basis of classification is variable and not fully specified in advance. Moreover, the program illustrates an important general point. Its success depends on the rules by which the continuous message is divided into appropriate segments. Segmentation seems likely to be a primary problem in all mechanical pattern recognition, particularly in the recognition of speech, since the natural pauses in spoken language do not generally come between words. MAUDE handles the segmentation problems in terms of context, and this will often be appropriate. In other respects MAUDE does not provide an adequate basis for generalizing about pattern recognition. The patterns of Morse code are too easy, and the processing is rather specialized.

Our second example deals with a more challenging problem: the recognition of hand-printed letters of the alphabet. The characters that people print in the ordinary course of filling out forms and questionnaires are surprisingly varied. Gaps abound where continuous lines might be expected; curves and sharp angles appear interchangeably; there is almost every imaginable distortion of slant, shape and size. Even human readers cannot always identify such characters; their error rate is about 3 per cent on randomly selected letters and numbers, seen out of context.

The first step in designing a mechanical reader is to provide it with a means of assimilating the visual data. By nature computers consider information in strings of bits: sequences of zeros and ones recorded in on-off devices. The simplest way to encode a character into such a sequence is to convert it into a sort of halftone by splitting it into a mesh or matrix of squares as fine as may be necessary. Each square is then either black or white—a binary situation that the machine is designed to handle. Making such halftones presents no problem. For example, an image of the letter could be projected on a bank of photocells, with the output of each cell controlling a binary device in the computer. In the experiments to be described here the appropriate digital information from the matrix was recorded on punch cards and was fed into the computer in this form.

Once this sequence of bits has been put in, how shall the program proceed to identify it? Perhaps the most obvious approach is a simple matching scheme, which would evaluate the similarity of the unknown to a series of ideal templates of all the letters, previously stored in digital form in the machine. The sequence of zeros and ones representing the unknown letter would be compared to each template sequence, and the number of matching digits recorded in each case. The highest number of matches would identify the letter.

In its primitive form the scheme would clearly fail. Even if the unknown were identical to the template, slight changes in position, orientation or size could destroy the match completely. (See Fig. 3*a*.) This difficulty has long been recognized, and in some character-recognition programs it has been met by inserting a level of information-processing ahead of the template-matching procedure. The sample is shifted, rotated and magnified or reduced in order to put it into a standard, or at least a more tractable, form.

Although obviously an improvement over raw matching, such a procedure is still inadequate. What it does is to compare shapes rather successfully. But letters are a good deal more than mere shapes. Even when a sample has been converted to standard size, position and orientation, it may match a wrong template more closely than it matches the right one. (See Fig. 3*b*.)

Nevertheless the scheme illustrates what we believe to be an important general principle. The critical change was from a program with a single level of operation to a program with two distinctly different levels. The first level shifts, and the second one matches. Such a hierarchical structure is forced on the recognition system by the nature of the entities to be recognized. The letter A is defined by the set of configurations that people call A, and their selections can be described—or imitated—only by a multilevel program.

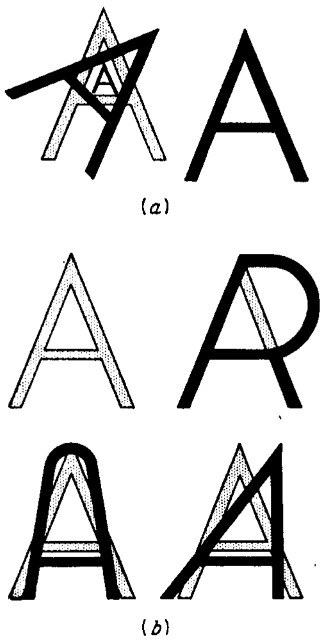


Figure 3. (*a*) Template matching cannot succeed when the unknown letter (color) has the wrong size, orientation, or position. The program must begin by adjusting sample to standard form. (*b*) Incorrect match may result even when sample (gray) has been converted to standard form. Here R matches A template more closely than do samples of the correct letter.

We have said that letter patterns cannot be described merely as shapes. It appears that they can be specified only in terms of a preponderance of certain *features*. Thus A tends to be thinner at the top than at the bottom; it is roughly concave at the bottom; it usually has two main strokes more vertical than horizontal, one more horizontal than vertical, and so on. All these features taken together characterize A rather more closely than they characterize any other letter. Singly none of them is sufficient. For example, W is also roughly concave at the bottom, and H has a pattern of horizontal and vertical strokes similar to that described for A. Each letter has its own set of probable features, and a successful character recognizer will determine which set is the best fit to an unknown sample.

So far nothing has been said about how the features are to be determined and how the program will use them. The template-matching scheme represents one approach. Its "features," in a sense, are the individual cells of the matrix representing the unknown sample, and its procedure is to match them with corresponding cells in the template. Both features and procedures are determined by the designer. We have seen that this scheme will not succeed. In fact, any system must fail if it tries to specify every detail of a procedure for identifying patterns that are themselves defined only ostensively. A pattern-recognition system must learn. But how much?

At one extreme there have been attempts to make it learn, or generate, everything: the features, the processing, the decision procedure. The initial state of such a system is called a "random net." A large number of on-off computer elements are multiply interconnected in a random way. Each is thus fed by several others. The thresholds of the elements (the number of signals that must be received before the element fires) are then adjusted on the basis of performance. In other words, the system learns by reinforcing some pathways through the net and weakening others.

How far a random net can evolve is controversial. Probably a net can come to act as though it used templates. However, none has yet been shown capable of generating features more sophisticated than those based, like templates, on single matrix cells. Indeed, we do not believe that this is possible.

At present the only way the machine can get an adequate set of features is from a human programmer. The effectiveness of any particular set can be demonstrated only by experiment. In general there is probably safety in numbers. The designer will do well to include all the features he can think of that might plausibly be useful.

A program that does not develop its own features may nevertheless be capable of modifying some subsequent level of the decision procedure, as we shall see. First however, let us consider that procedure itself. There are two fundamentally different possibilities: sequential and parallel processing. In sequential processing the features are inspected in a predetermined order, the outcome of each test determining the next step. Each letter is represented by a unique sequence of binary decisions. To take a simple example, a program to distinguish the letters A, H, V and Y might decide among them on the basis of the presence or absence of three features: a concavity at the top, a crossbar and a vertical line. The sequential process would ask first: "Is there a concavity at the top?" If the answer is no, the sample is A. If the answer is yes, the program asks: "Is there a crossbar?" If yes, the letter is H; if no, then: "Is there a vertical line?" If yes, the letter is Y; if no, V. (See Fig. 4.)

In parallel processing all the questions would be asked at once, and all the answers presented simultaneously to the decision maker. (See Fig. 5.) Different combinations identify the different letters. One might think of the various features as being inspected by little demons, all of whom then shout the answers in concert to a decision-making demon. From this conceit comes the name "Pandemonium" for parallel processing.

Of the two systems the sequential type is the more natural for a machine. Computer programs are sequences of instructions, in which choices or alternatives are usually introduced as "conditional transfers": Follow one set of instructions if a certain number is negative (say) and another set of instructions if it is not. Programs of this kind can be highly efficient, especially in cases where any given decision is almost certain to be right. But in "noisy" situations sequential programs require elaborate checking and backtracking procedures to compensate for erroneous decisions.

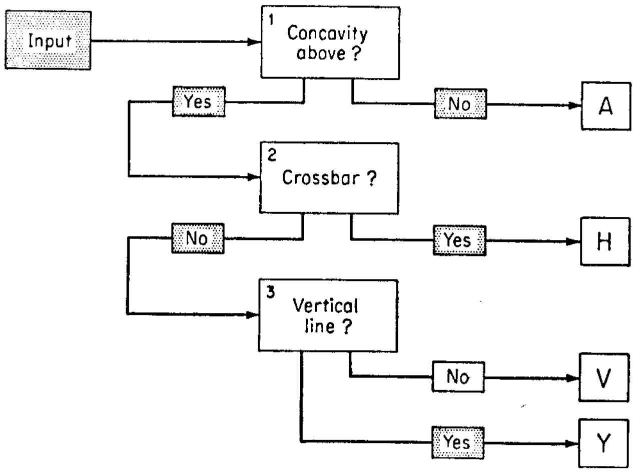


Figure 4. Sequential-processing program for distinguishing four letters, A, H, V and Y, employs three test features: presence or absence of a concavity above, a crossbar, and a vertical line. The tests are applied in order, with each outcome determining the next step.

Parallel processing, on the other hand, need make no special allowance for error and uncertainty.

Furthermore, some features are simply not subject to a reasonable dichotomy. An A very surely has a crossbar, an O very surely has not. But what about B? The most we can say is that it has more of a crossbar than O, and less than A. A Pandemonium program can handle the situation by having the demons shout more or less loudly. In other words, the information flowing through the system need not be binary; it can represent the quantitative preponderance of the various features.

Still another advantage of parallel processing lies in the possibility of making small changes in a network for experimental purposes. In typical sequential programs the only possible changes involve replacing a zero with a one, or vice versa. In parallel ones, on the other hand, the weight given to crossbarness in deciding if the unknown is actually B may be changed by as small an amount as desired. Experimental changes of this kind need not be made by the programmer alone. A program can be designed to alter internal weights as a result of experience and to profit from its mistakes. Such learning is much easier to incorporate into a Pandemonium than into a sequential system, where a change at any point has grave consequences for large parts of the system.

Parallel processing seems to be the human way of handling pattern

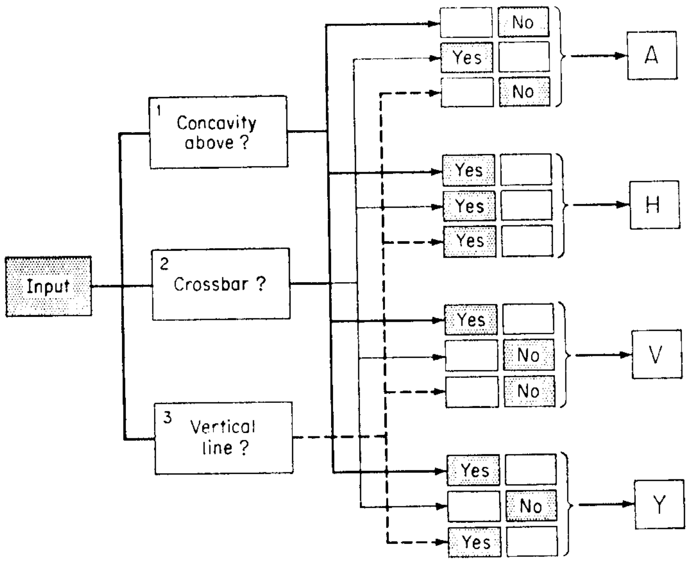


Figure 5. Parallel-processing program uses the same test features as the sequential program in Fig. 4, but applies all tests simultaneously and makes decision on the basis of the combined outcomes. The input is a sample of one of the letters A, H, V and Y.

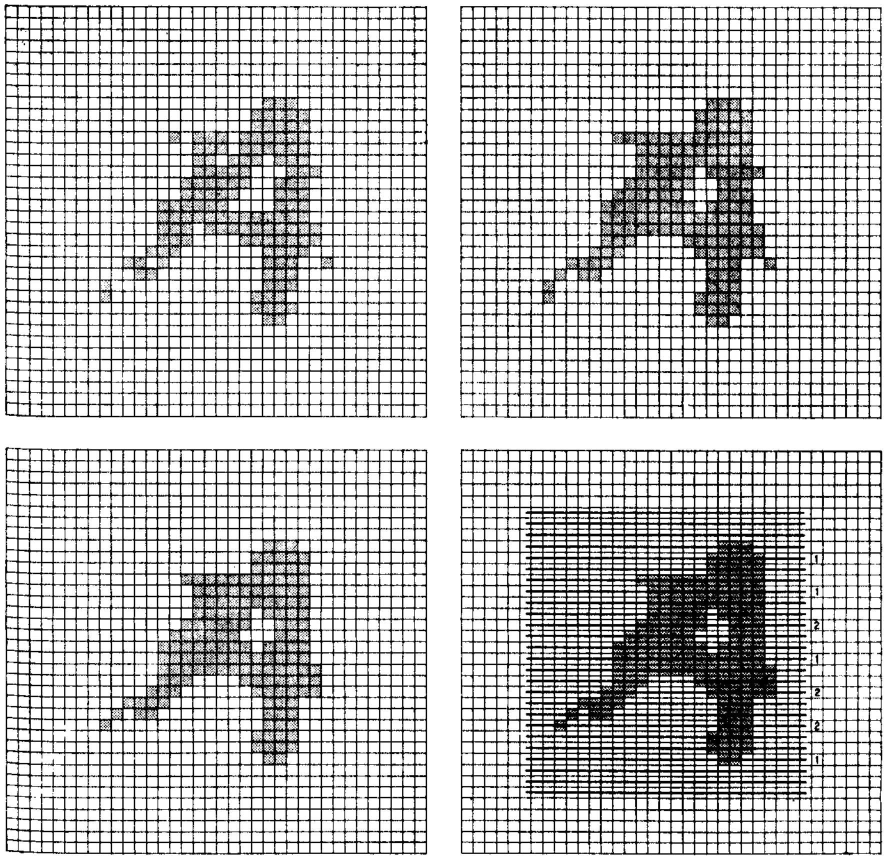


Figure 6. Hand-printed letter A is processed for recognition by a computer. Original sample is placed on grid and converted to a cellular pattern by completely filling in all squares through which lines pass (top left). The computer then cleans up the sample, filling in gaps (top right) and eliminating isolated cells (botton left). The program tests the pattern for a variety of features. The test illustrated here (bottom right) is for the maximum number of intersections of the sample with all horizontal lines across the grid.

recognition as well. Speech can be understood if all acoustic frequencies above 2000 cycles per second are eliminated, but it can also be understood if those below 2000 are eliminated instead. Depth perception is excellent if both eyes are open and the head is held still; it is also excellent if one eye is open and the head is allowed to move.

A Pandemonium system that learns from experience has been tested by Worthie Doyle of the Lincoln Laboratory. At present it is programmed to identify 10 hand-printed characters, and has been tested on samples of A, E, I, L, M, N, O, R, S and T. The program has six levels: (1) input,

Type of test and designation		Outcome	A	E	I	L	M	N	O	R	S	T
Horizontal and vertical cross-sections	HOMSXC	3	.083	.070			.250	.347	.097	.056		.097
	VEMSXC	3	.073	.339			.040		.008	.194	.258	.089
	HORUNS	2111111		.500						.500		
	VERUNS	2111111						1.000				
Strokes	HORSTR	1	.182	.006	.125	.125	.125	.146	.016	.057	.016	.203
	VERSTR	2	.178	.007			.170	.207	.229	.207		
Edge lengths and ratios	SEDGE	1	.267	.007		.014	.158	.115	.007	.165		.266
	WEDGE	1	.083	.071	.024	.024	.035	.012		.047	.318	.389
	NEDGE	2	.259	.024	.153	.024	.106	.106	.071	.059	.189	.012
	EEDGE	4	.232		.161		.214	.286	.107			
	NO:SOU	4	.513				.205	.077		.128		.077
	EA:WES	1	.055	.400		.309	.018	.036		.163		.018
Profiles	SCUCAV	3	.150				.800	.050				
	WESCAV	2	.047	.094	.023	.012	.023	.035	.035	.059	.412	.259
	NORCAV	1	.133	.177	.100	.092	.004		.133	.108	.116	.137
	EASCAV	1	.155	.005	.115	.095	.105	.130	.170	.010	.050	.165
	SOUBOT	220	.268	.106		.068	.159	.167	.008	.220	.008	
	WESBOT	221	.030	.030	.061						.364	.515
	NORBOT	121	.290	.145					.354	.042	.042	.125
	EASBOT	121	.326				.020	.102	.266	.020	.245	.020
Internal structure	SBOTSG	2	.250	.008		.016	.125	.141	.219	.203	.039	
	WBOTSG	1	.161	.076	.090	.099	.108	.121	.063	.081	.045	.157
	NBOTSG	1	.119	.190	.111	.102	.013	.018	.089	.040	.159	.159
	EBOTSG	1	.147	.058	.098	.103	.103	.121	.062	.071	.076	.061
	SOUBEN	20					.333	.167				.500
	WESBEN	10	.198	.143	.011	.022	.121	.132	.011	.099	.022	.241
	NORBEN	10	.169	.180		.135	.079			.146	.247	.045
	EASBEN	10	.211	.012	.012	.118	.176	.106		.176		.188
Total score			4.579	2.648	1.084	1.358	3.490	3.622	1.945	2.851	2.606	3.823

Figure 7. Recognition program for hand-printed letters applies the 28 feature tests listed by code name at left. Names represent such features as maximum intersection with horizontal line (HOMSXC), concavity facing south (SOUCAV), and so on. Figures in right-hand section are relative probabilities of all letters for each test outcome. The program decides on the letter with the largest total of all probabilities. In the example shown here the decision is for the letter A, with a probability total of 4.579.

(2) cleanup, (3) inspection of features, (4) comparison with learned-feature distribution, (5) computation of probabilities and (6) decision. The input is a 1024-cell matrix, 32 on a side. At the second level the sample character is smoothed by filling in isolated gaps and eliminating isolated patches. (See Fig. 6.)

Recognition is based on such features as the relative length of different edges and the maximum number of intersections of the sample with a horizontal line. (The computer "draws" the lines by inspecting every horizontal row in the matrix, and recognizes "intersections" as sequences of ones separated by sequences of zeros.) No single feature is essential to recognition, and various numbers of them have been tried. The particular program shown here uses 28. (See Fig. 7.)

Every letter fed into the machine is tested for each of the features. During the learning phase a number of samples of each of the 10 letters is presented and identified. For every feature the program compiles a table or "census." It tests each sample and enters the outcome under the appropriate letter. When the learning period is finished, the table shows how many times each outcome occurred for each of the 10 letters. Figure 8, which refers to maximum intersections with a horizontal line, represents the experience gained from a total of 330 training samples. It shows, for example, that the outcome (three intersections) occurred 72 times distributed among six A's, five E's, 18 M's, 25 N's, seven O's, four R's, seven T's and no other letters. The other possible outcomes are similarly recorded.

Next the 28 censuses are converted to tables of estimated probabilities,

Letter	Samples	Outcome			
		1	2	3	4
A	39		33	6	
E	46	6	35	5	
I	25	25			
L	24	7	17		
M	24			18	6
N	28		2	25	1
O	34		27	7	
R	33		28	4	1
S	38	8	30		
T	39	10	22	7	
Total	330	56	194	72	8

Figure 8. "CENSUS" represents information learned by letter-recognition program during training period. This table summarizes the outcomes of the test for maximum number of intersections with a horizontal line, applied to a total of 330 identified samples in the learning process.

by dividing each entry by the appropriate total. Thus the outcome—three intersections—comes from an A with a probability of .083 (6/72); an E, with a probability of .070 (5/72), and so on.

Now the system is ready to consider an unknown sample. It carries out the 28 tests and "looks up" each outcome in the corresponding feature census, entering the estimated probabilities in a table. Then the total probabilities are computed for each letter. The final decision is made by choosing the letter with the highest probability.

This program makes only about 10 per cent fewer correct identifications than human readers make—a respectable performance, to be sure. At the same time, the things it cannot do point to the difficulties that still lie ahead. We would emphasize three general problems: segmentation, hierarchical learning and feature generation.

Characters must be fed in one at a time. The program is unable to segment continuous written material. The problem will doubtless be relatively easy to solve for text consisting of separate printed characters, but will be more formidable in the case of cursive script.

The program learns on one level only. The relation between feature presence and character probability is determined by experience; everything else is fixed by the designer. It would certainly be desirable for a character recognizer to use experience for more general improvements: to change its cleanup procedures, alter the way probabilities are combined and refine its decision process. Eventually we look to recognition of words; at this point the program will have to learn a vocabulary so that it can use context in identifying dubious letters. At the moment, however, neither we nor any other designers have any experience with the interaction of several levels of learning.

The most important learning process of all is still untouched: No current program can generate test features of its own. The effectiveness of all of them is forever restricted by the ingenuity or arbitrariness of their programmers. We can barely guess how this restriction might be overcome. Until it is, "artificial intelligence" will remain tainted with artifice.

A PATTERN-RECOGNITION PROGRAM THAT GENERATES, EVALUATES, AND ADJUSTS ITS OWN OPERATORS

by Leonard Uhr & Charles Vossler

Background Review

The typical pattern-recognition program is either elaborately preprogrammed to process specific arrays of input patterns, or else it has been designed as a *tabula rasa,* with certain abilities to adjust its values, or "learn." The first type often cannot identify large classes of patterns that appear only trivially different to the human eye, but that would completely escape the machine's logic (Bailey and Norrie, 1957; Greanias et al., 1957). The best examples of this type are probably capable of being extended to process new classes of patterns (Grimsdale et al., 1959*a*; Sherman, 1959). But each such extension would seem to be an *ad hoc* complication where it should be a simplification, and to represent an additional burden of time and energy on both programmer and computer.

The latter type of self-adjusting program does not, at least as yet, appear to possess methods for accumulating experience that are sufficiently powerful to succeed in interesting cases. The random machines show relatively poor identification ability (Rosenblatt, 1958, 1960*a*). (One exception to this statement appears to be Roberts' modification of Rosenblatt's Perceptron (Roberts, 1960). But this modification appears to make the Perceptron an essentially nonrandom computer.) The most successful of this type of computer, to date, simply accumulates information or probabilities about discrete cells in the input matrix (Baran and Estrin, 1960; Highleyman and Kamentsky, 1960). But this is an unusually weak type of learning (if it should be characterized by that vague epithet at all), and this type of program is bound to fail as soon as, and to the extent that, patterns are allowed to vary.

Several programs compromise by making use of some of the self-adapting and separate operator processing features of the latter type of program, but with powerful built-in operations of the sort used by the first type (Doyle, 1960; Unger, 1959). They appear to have gained in flexibility in writing and modifying programs; but they have not, as yet, given (published) results that indicate that they are any more powerful than the weaker sort of program (*e.g.*, Baran and Estrin) that uses individual cells in the matrix in ways equivalent to their use of "demons" and "operators." A final example of this mixed type of program is the randomly coupled "*n*-tuple" operator used by Bledsoe and Browning (1959; 1961*a*). In this program, random choice of pairs, quintuples and other tuples of cells in the input matrix is used to compose operators, in an attempt to get around the problems of preanalyzing and preprogramming. This method appears to be guaranteed to have at least as great power as the single cell probability method (Uhr, 1961*b*). But it has not as yet demonstrated this power. And it would, like most of the other programs discussed (or known to the authors) fall down when asked to process patterns which differed very greatly from those with which it had originally "gained experience" by extracting information (Uhr, 1960).

Summary of Program Operation

In summary, the original running pattern recognition program works as follows: Unknown patterns are presented to the computer in discrete form, as a 20×20 matrix of zeros and ones. The program generates and composes operators by one of several random methods, and uses this set of operators to transform the unknown input matrix into a list of characteristics. Or, alternately, the programmer can specify a set of pregenerated operators in which he is interested.

These characteristics are then compared with lists of characteristics in memory, one for each type of pattern previously processed. As a result of similarity tests, the name of the list most similar to the list of characteristics just computed is chosen as the name of the input pattern. The characteristics are then examined by the program and, depending on whether they individually contributed to success or failure in identifying the input, amplifiers for each of these characteristics are then turned up or down. This adjustment of amplifiers leads eventually to discarding operators which produce poor characteristics, as indicated by low amplifier settings, and to their replacement by newly generated operators.

One mode of operation of the present program is to begin with no operators at all. In this case operators are initially generated by the program at a fixed rate until some maximum number of operators is reached. The continual replacement of poor operators by new ones then tends to produce an optimum set of operators for processing the given array of inputs.

Details of Program Operation

The program can be run in a number of ways, and we will present results for some of these. The details of the operation of the program follow.

1. An unknown pattern to be identified is digitized into a 20 × 20 0–1 input matrix (Fig. 1).
2. A rectangular mask is drawn around the input (its sides defined by the leftmost, rightmost, bottommost, and topmost filled cells) (Fig. 2).
3. The input pattern is transformed into four 3-bit characteristics by each of a set of 5 × 5 matrix operators, each cell of which may be visualized as containing either a 0, 1, or blank. These small matrices which measure local characteristics of the pattern are translated, one at a time, across and then down that part of the matrix which lies within the mask. The operator is considered to match the input matrix whenever the 0's and 1's in the operator correspond to identical values in the pattern, and for each match the location of the center cell of the 5 × 5 matrix operator is temporarily recorded (Fig. 3). This information is then summarized and scaled from 0 to 7 to form four 3-bit characteristics for the operator. These represent (1) the number of matches, (2) the average horizontal position of the matches within the rectangular mask, (3) the average

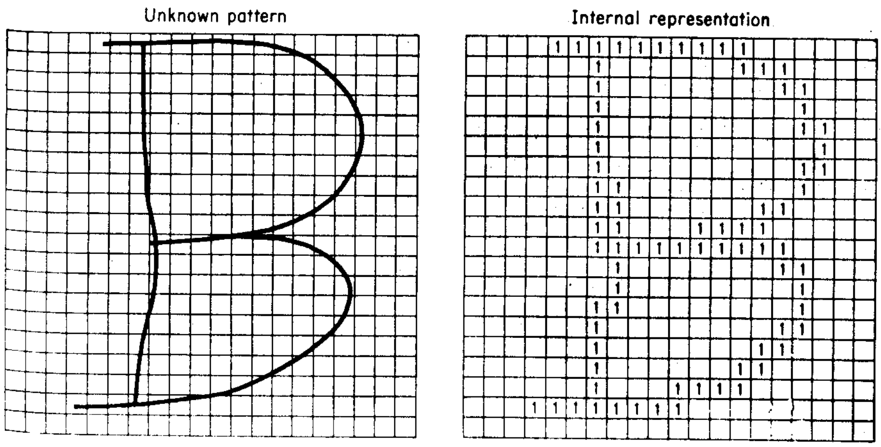


Figure 1. An unknown pattern is input as a 20 × 20 matrix with the cells covered by the pattern represented by "1's" and the other cells by "0's."

vertical position of the matches, and (4) the average value of the square of the radial distance from the center of the mask.

A variable number of operators can be used in any machine run. This can mean either a number preset for that specific run, or a number that begins at zero and expands, under one of the rules described below, up to the maximum. The string of 25 numbers which defines a 5 × 5 matrix operator can be generated in any of the following ways (Fig. 4):

a. A preprogrammed string can be fed in by the experimenter.
b. A random string can be generated; this string can be restricted as to the number of "ones" it will contain, and as to whether these "ones" must be connected in the 5 × 5 matrix. (We have not actually tested this method as yet.)
c. A random string can be "extracted" from the present input matrix and modified by the following procedure (which in effect is imitating a certain part of the matrix). The process of inserting blanks

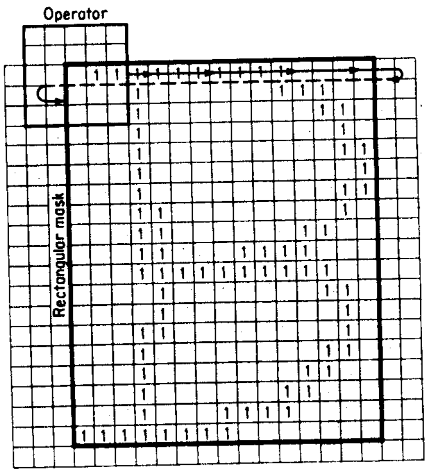
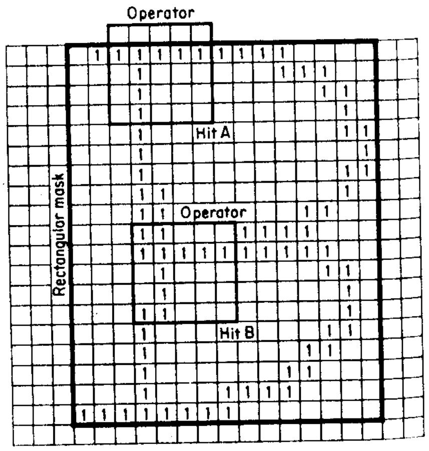
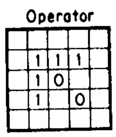


Figure 2. A rectangular mask is drawn around the unknown pattern. Each of the 5 × 5 matrix "operators" is then translated over the pattern.





Hit	X	Y	R^2
A	1	0	4
B	2	4	0
N = 2	2	2	2

Figure 3. The operator at the lower left in the figure is shown in the two positions where it matches the input matrix. An operator gives a positive output each time its "1's" cover "1's" and its "0's" cover "0's" in the unknown pattern.

in the extracted operator allows for minor distortions in the local characteristics which the operator matches.

(1) A 5 $\times$ 5 matrix is extracted from a random position in the input matrix.
(2) All "zero" cells connected to "one" cells are then replaced by blanks.
(3) Each of the remaining cells, both "zeros" and "ones," is then replaced by a blank with a probability of ½.
(4) Tests are made to ensure that the operator does not have "ones" in the same cells as any other currently used operator or any operator in a list of those recently rejected by the program. If the operator is similar to one of these in this

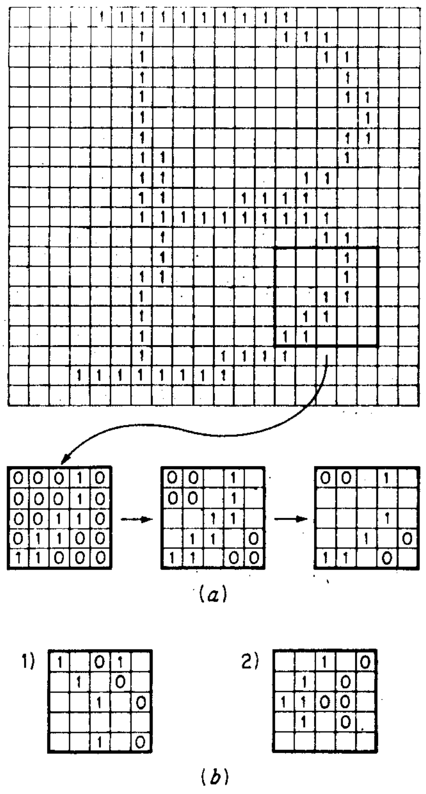


Figure 4. Operators are generated within the 5 $\times$ 5 matrix by either: (*a*) extraction from the input pattern (random placement of a 5 $\times$ 5 matrix, elimination of "0's" connected to "1's" and elimination of each of the remaining cells with a probability of ½) or (*b*) by random designation of cells as either "0" or "1" (choose a "1," then place a "0" two cells to its right). In 1) from 3 to 7 "1's" are chosen completely at random, while in 2) the choice is limited to connected cells.

respect a new operator is generated by starting over at step 1 (Fig. 5).

4. A second type of operator is also used. This is a combinatorial operator which specifies one of 16 possible logical or arithmetic operations and two previously calculated characteristics which are to be combined to produce a third characteristic. These operators are generated by the program by randomly choosing one of the possible operations and the two characteristics which are to be combined. This random generation process is improved by generating a set of ten operators, and then pretesting these using the last two examples of each pattern which have been saved in memory for this purpose. This pretesting is designed to choose an operator from the set which produces characteristics that tend to be invariant over examples of the same pattern yet vary between different patterns.

Since these operators may act upon characteristics produced by previous operators of the same type, functions of considerable complexity may be built up.

5. The two types of operators just described produce a list of characteristics by which the program attempts to recognize the unknown input

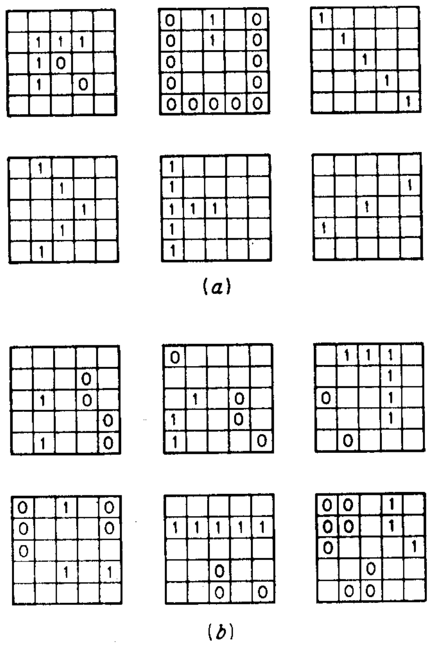


Figure 5. (*a*) Some typical examples of preprogrammed operators are shown. (*b*) Six of the operators generated by the program, during a run that reached 94 per cent success on 7 sets of 5 patterns, are shown.

pattern (Fig. 6). At any time the program has stored in memory a similar list of characteristics for each type of pattern which the program has previously encountered. Corresponding to each list of characteristics in memory is a list of 3-bit amplifiers, which gives the current weighting for each characteristic as a number from 0 to 7.

The recognition process proceeds by taking the difference between each of the characteristics for the input pattern and those in the recognition list of the first pattern. These differences are then weighted by the corresponding pattern amplifiers, and then by general amplifiers which represent the average of the pattern amplifiers across all patterns, producing a weighted average difference between the input list and the list in memory. This average difference is multiplied by a final "average difference" amplifier to obtain a "difference score" for the list in memory. When a difference score has been computed for each list in memory, the name of the list with the smallest score is printed as the name of the input pattern (Fig. 7).

6. After each pattern is recognized the program modifies pattern amplifiers in those patterns which have difference scores less than or only slightly above the difference score for the correct pattern (Fig. 8). This means that the program will tend to concentrate on the difficult discrimination problems, since amplifiers are adjusted only in those patterns which appear similar to the correct pattern in terms of the difference scores and therefore make identification of the input difficult. The correct pattern is compared with each of the similar patterns in turn. Each characteristic in the memory lists for a pair of patterns is examined individually, and a determination is made as to whether the correct pattern would have been chosen if the choice had been made on the basis of this characteristic alone. If this one characteristic would have identified the correct pattern, then the corresponding amplifier is turned up by one. If it would have identified the wrong pattern then the amplifier is turned down by one. If no information is given by the characteristic, for example, if it is the same

	Matrix operators									Combinatorial operators		
Pattern	Operator 1				Operator 2					Characteristic		
Name	N	X	Y	R^2	N	X	Y	R^2	...	m-2	m-1	m
?	2	2	2	2	2	3	1	6	....	6	7	4
A	3	3	4	1	4	0	1	1	...	1	2	3
B	2	2	3	2	3	3	2	5	...	4	7	5
C	4	5	6	5	1	0	0	4	...	2	1	7

Figure 6. Operator outputs are listed for the unknown pattern in the same format as in lists stored in memory.

for both patterns, then the amplifier is turned down with a probability of ⅛. If the pattern compared with the correct pattern had the higher difference score then the amplifiers are adjusted only in that pattern. Otherwise, amplifiers are adjusted in both patterns. This means that if several patterns obtained lower scores than the correct pattern then the amplifiers in the correct pattern will be drastically changed, since they will change when compared with each of these patterns.

The list of characteristics in memory for the pattern just processed is then modified. The first time a pattern is encountered its list of computed characteristics is simply stored in memory along with its name. On the second encounter of a pattern each of the characteristics in memory is replaced by the new characteristic with a probability of ½. For the third and following encounters each characteristic is replaced by the new value with a probability of ¼. Since about ¼ of the characteristics will be changing each time, after several examples of a pattern have been processed, the list of characteristics in memory will tend to be more similar to the characteristics of the last patterns processed than to those processed

PATTERN *A*

Characteristics (A)	:	3	3	4	1	4	. . .	3
Input (?)	:	2	2	2	2	2	. . .	4
Difference |A − ?|	:	1	1	2	1	2	. . .	1
Pattern amplifiers	:	2	3	1	2	0	. . .	3
General amplifiers	:	3	3	1	1	0	. . .	3
Diff. X amplifiers	:	6	9	2	2	0	. . .	9

Weighted av. diff.	Av. diff. amplifier	Diff. score
$\frac{28}{27} = 1.04$	61	63

PATTERN *B*

Characteristics (B)	:	2	2	3	2	3	. . .	5
Input (?)	:	2	2	2	2	2	. . .	4
Difference |B − ?|	:	0	0	1	0	1	. . .	1
Pattern amplifiers	:	4	3	2	3	2	. . .	2
General amplifiers	:	3	3	1	1	0	. . .	3
Diff. X amplifiers	:	0	0	2	0	0	. . .	6

Weighted av. diff.	Av. diff. amplifier	Diff. score
$\frac{8}{32} = 0.25$	60	15

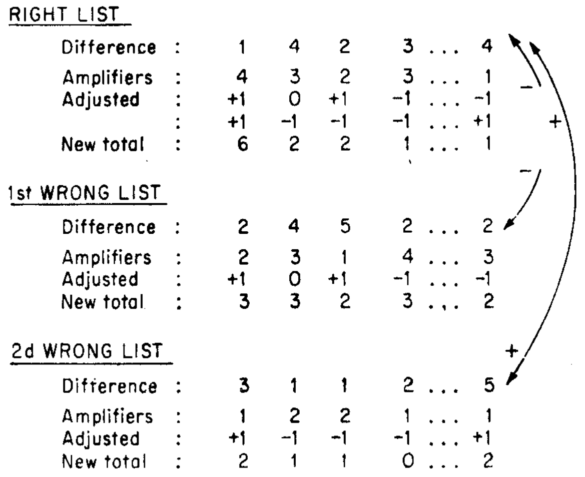
Figure 7. Differences are obtained between the characteristics for the input pattern and each list of characteristics in memory. These differences are then weighted by the product of the "general amplifiers" and "pattern amplifiers," giving a weighted average difference for each list in memory. When multiplied by corresponding "average difference amplifiers," the weighted average differences give "difference scores" for each pattern in memory. The name of the pattern with the smallest "difference score" is chosen as the name of the input.

earlier. However, to the extent that the learning process is able to produce operators giving invariant characteristics for a single pattern, the list of characteristics will be representative of all the examples processed. The reason for not simply using the average value for each characteristic is that this would require saving in memory more than the 3 bits otherwise needed for each characteristic, as well as saving an indication of the number of times each characteristic had been calculated for each pattern.

An alternate scheme which we tried involved saving the highest and lowest values obtained by each characteristic, and averaging these to obtain a mean value with which to compare the input. This worked quite well in all our test runs, which used a few samples of each pattern. But there is the possibility that with large numbers of examples of a pattern, all the characteristics will eventually have very large ranges; that is, the lower bounds will tend to be 0 and the upper bounds will tend to be 7.

7. The average difference amplifiers which are used in the final step of the recognition process provide only coarse adjustments. These amplifiers are initially set to some fixed value, *e.g.*, 60, and are then adjusted for the same pairs of patterns as the pattern amplifiers. The amplifier for the correct pattern is turned down by N if there are N incorrect patterns, and the amplifier for each of the similar patterns is turned up by one.

8. The general characteristic amplifiers are now computed by averaging the pattern amplifiers across all patterns. These indicate the general value of each characteristic in the recognition process and form the basis for the construction of success counts which control the replacement of operators. Since the combinatorial operators combine characteristics to produce other characteristics, the success count should reflect both the value of a charac-



RIGHT LIST

Difference	:	1	4	2	3 ...	4
Amplifiers	:	4	3	2	3 ...	1
Adjusted	:	+1	0	+1	−1 ...	−1
	:	+1	−1	−1	−1 ...	+1
New total	:	6	2	2	1 ...	1

1st WRONG LIST

Difference	:	2	4	5	2 ...	2
Amplifiers	:	2	3	1	4 ...	3
Adjusted	:	+1	0	+1	−1 ...	−1
New total	:	3	3	2	3 ...	2

2d WRONG LIST

Difference	:	3	1	1	2 ...	5
Amplifiers	:	1	2	2	1 ...	1
Adjusted	:	+1	−1	−1	−1 ...	+1
New total	:	2	1	1	0 ...	2

Figure 8. The pattern amplifiers for certain lists are adjusted to *increase* weightings of individual characteristics that gave differences in the right direction, and to *decrease* weightings that gave differences in the wrong direction.

teristic in the recognition process and the importance of this characteristic in aiding the creation of other, possibly important characteristics.

9. This success count is formed by first storing the value of the general characteristic amplifier corresponding to each characteristic in a table for success counts. Then starting with the last combinatorial operator and working back through the list of these operators, ½ the value of the success count for the characteristic corresponding to the operator is added to the success counts of the two characteristics which the operator combines. Finally, two times the general characteristic amplifier setting is added to each success count.

10. Whenever a new operator is generated, the characteristics produced by the operator are computed for each of the possible patterns using the last example of each pattern, which has been saved in the computer memory. These newly calculated characteristics are then inserted into the list of characteristics for their respective patterns. At the same time the pattern amplifier settings for each of these new characteristics are set to 1 so that the characteristic will have very little weight in computing a difference score until it has been turned up as a function of proved ability at differentiation. Since the general amplifier for a characteristic is simply the average of the pattern amplifiers, it will also be 1 for the new characteristic. The success count of a new characteristic which is not combined to produce other characteristics is then 3 and this value will tend to increase if the operator proves to be valuable. On the other hand if a success count drops below 3 (or in the case of a matrix operator, if the average value of the success counts of its four characteristics drops below 3) the operator is rejected and a new operator is generated to take its place.

The pattern amplifiers play a crucial part both by aiding directly in the recognition process and by providing the information which ultimately determines the generation of new operators to replace poor ones. Since the adjustment of these amplifiers is made selectively, based on their individual success or failure in distinguishing pairs of patterns where confusion is likely, the operators rejected by the program will tend to be those which are not useful in making the more difficult discrimination. Also, because amplifiers are usually changed more drastically when the computer makes an incorrect guess, the 5×5 matrix operators will have a higher probability of being extracted from unrecognized patterns. Although the rules governing the learning process seem rather arbitrary in many cases, and it is difficult to describe their effects quantitatively, qualitative effects, such as this ability to concentrate on difficult problems, are fairly easy to show. The description of the program's operation shows that the emphasis is not so much on the design of a specific problem-solving code as it is on the design of a program which, at least in part, will construct such a problem-solving code as a result of experience.

It is interesting to note that the memory of the program exists in at least three different places: (1) in the lists of characteristics in memory, (2) in the settings of the various amplifiers, and (3) in the set of operators in use by the program. While the lists of characteristics bear some direct relationship to the individual patterns processed by the program, the values of the amplifiers and the set of operators in use by the program depend in a more complex way on the whole set of patterns processed by the program, and on the program's success or failure in recognizing these patterns. The learning in the first case, which involves simply storing characteristics in memory, is merely "memorization" or "learning by rote." In the second case, the learning is more subtle for it involves the program's own analysis of its ability to deal with its environment, and its attempt to improve this ability.

Test Results of Original Program

The original program was written for the IBM 709 and required about 2000 machine instructions. The time required to process a single character was about 25 seconds when 5 different patterns were used and 40 seconds when each character had to be compared with ten possible patterns in memory. While such times are not excessive, they are large enough to make it impractical to run extremely large test cases.

In several early runs which we made, 48 preprogrammed matrix operators were used. These were designed to measure such things as straight and curved lines, the ends of vertical and horizontal strokes, and various other features. The program was tested using seven different sets of the five hand-printed characters A, B, C, D, and E. These involved a fair amount of distortion, and variation in size, but were not rotated to any great extent.

The program's performance on the last three or four sets in a run varied from about 70% to 80% depending on various changes which were made to the rules governing the learning process.

One run was made using the individual cells of the input matrix as first level operators, building up higher-level combinatorial operators on these. This gave little better than 30% success. Finally, this version of the program was tested without any preprogrammed operators, the program achieved 97% on known and 70% on unknown examples of a ten-letter alphabet. It also showed ability to recognize simple drawings of objects.

Test Results of Revised Program

The original program was modified, chiefly to increase its running speed, and secondarily to simplify some of its logic. In order to make use of logical machine instructions on the 709, all characteristics and their ampli-

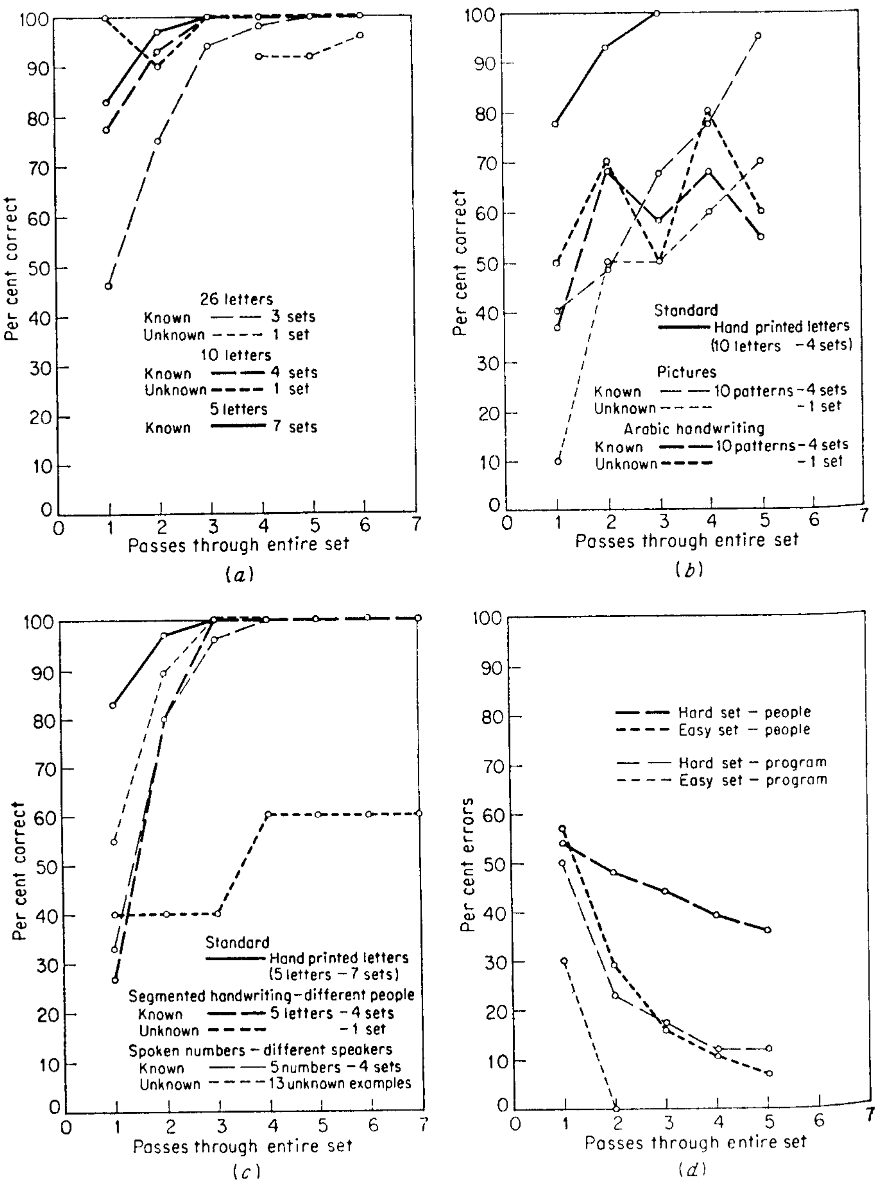


Figure 9. (*a*) Results of the computer simulation program. Hand-printed alphabetic patterns. Per cent correct on several sets of a 26-pattern, a 10-pattern, and a 5-pattern array. The program was tested on both known and unknown sets of patterns. (*b*) Results with two additional 10-pattern arrays: (1) line drawings of pictures (different examples of each of 5 faces and 5 objects), (2) arabic handwriting (written by the same person). The program was tested on both known

fiers were reduced to 1-bit values. The revised program stores nine one-bit values for each operator—whether it hit at least once in each of nine parts of the matrix. Operators are weighted either 1 (to be used) or 0 (not to be used), referring to the characteristics they give for each pattern stored in memory. Operators are eliminated when they have given wrong outputs on the last *n* example for which the program as a whole has been wrong. The "general amplifiers" have been eliminated. These changes effected an increase of speed by a factor of about 40. Thus this program takes about 1 second per example for a 10-pattern alphabet on the 709, and less than .2 second on the 7090. This program is probably weaker than the original program, since it has virtually no range within which to search for a good set of weightings for its operators. But its increased speed led to its use for the bulk of our tests on this version of the program.

The speeded up program has been tested on several different types of input patterns, as shown in Fig. 9. In most cases, results were quite similar on both "known" examples (that is, examples the program had previously processed and hence had learned from) and "unknown" examples (that is, different from the ones used in learning, and also produced by different people). Figure 9*a* shows results for several different sizes of pattern arrays, all of hand-printed capital letters, printed by different people. These results show relatively little decrease in the program's abilities as the array size is increased, at least up to the 26-letter alphabet. Thus, on the sixth pass through the 26-letter alphabet the program was 100% correct on known patterns and 96% correct on unknown patterns.

Figure 9*b* presents results for two 10-pattern arrays. These were: (1) line drawings of cartoon faces and simple objects (such as shoes and pliers), each copied from a different picture, as found in cartoon strips and mail-order catalogs (Fig. 10 presents some examples of the cartoons), and (2) handwritten arabic letters, written by the same person. The program achieved 95% success on known and 70% success on unknown pictures, and 60% success on known and 55% success on unknown arabic letters (segmented handwriting) in the fifth pass. Figure 9*c* presents results from two 5-pattern arrays: (1) digitized and degraded sound spectrograms of speech (the numbers "zero," "one," "two," "three," and "four," as spoken by different speakers) (Uhr and Vossler, 1961*d*), and (2) segmented lower-case handwriting, written by different people. The

and unknown sets of patterns. (*c*) Results with two additional 5-pattern arrays: (i) spoken numerals (spoken by different people), (ii) segmented handwriting (written by different people). The program was tested on both known and unknown sets of patterns. (*d*) Results from a comparison experiment. Per cent errors for the program and mean per cent errors for human subjects (from 6 to 10 subjects per point) on one hard and one easy set of "meaningless" patterns. Both sets contained five variant examples of each of five patterns.

program achieved 100% success on both known and unknown spoken numerals by the fourth pass, and 100% success on known handwriting by the third pass. It achieved 60% success on the unknown handwriting, but it is likely that it would have improved further on these inputs if it had been given more opportunity to learn (once it achieves 100% success on known patterns it does not benefit appreciably from subsequent learning experiences).

Finally, the program's performance was compared with the performance of human subjects on sets of "meaningless" patterns. This sort of pattern minimizes the effects of the human being's lifetime of experience and resulting associative context. Figure 9*d* presents results from two such experiments, in both of which the program performed appreciably better than did any of the human subjects. Three additional experiments previously reported in the psychological literature were replicated. In all cases the program performed at a higher level than did the human subjects (Uhr, Vossler, and Uleman, 1962).

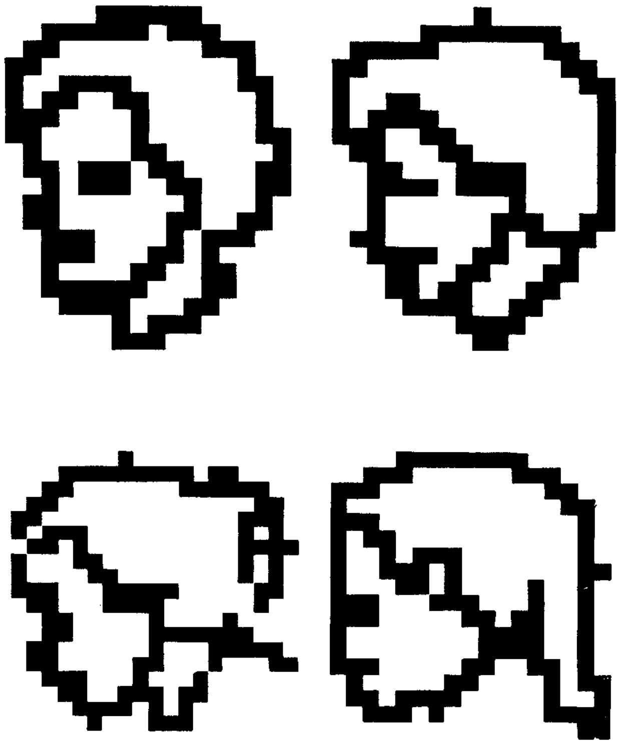


Figure 10. Two examples of each of two cartoon faces as presented to the simulation program (digitized by hand into a 20 × 20 matrix, after optical projection from the newspaper original).

This program was also tested for its ability to handle continuous patterns such as handwritten words and spoken sentences. Simple additional subroutines were written to allow it to input matrices any number of columns long, to make very primitive tentative segmentations (in every *n*th column, *n* usually around 7, and in columns with fewer than two filled cells), and to decide among the various alternatives at the various different tentative segmentation points. The program reached 100% correct performance when asked to segment and recognize the words, or alternatively the phonemes, in the simple sentence "Did Dad say before," spoken by different people.

On the handwritten sequences "pattern one," "pattern two," "pattern three" (written by different people), the program reached about 60% success in recognizing the letters. These tests do not in any adequate way sample the range of problems to be encountered with such stimuli. But they give some indication that the program is capable of at least beginning to handle continuous inputs. And it should be relatively easy to improve upon this performance by adding more sophisticated segmenting methods and a straightforward method (such as the use of letter *n*-tuple frequencies in the language) for making use of contextual information.

Discussion

When this program is given a neurophysiological interpretation, or a neural, net analog, it can be seen to embody relatively weak, plausible, and "natural-looking" assumptions. The 5×5 matrix operator is equivalent to a 5×5 net of input retinal cones or photocells converging on a single output, with "ones" denoting excitatory and "zeros" denoting inhibitory connections, and the threshold for firing the output unit set at the sum of the "ones." Each translation step of the operator matrix over the larger matrix gives a sequential simulation of the parallel placement of many of these simple neural net operators throughout the matrix. Each different operator, then, is the equivalent of an additional connection pattern between input cones, firing onto a new output unit that computes the output for that operation. This is all quite plausible for the retina as known anatomically, with a single matrix of cones in parallel that feed into several layers of neurons. Evidence for excitatory and inhibitory connections is also strong (Hartline, 1938). And there is even beginning to be evidence of several types of simple net operators that exist in parallel iterated form throughout the retinal matrix [four of these as determined by Lettvin, Maturana, McCulloch, and Pitts in the frog (1959); and probably even more as determined by Hubel and Wiesel in the cat (1959)].

It would seem, however, that the known physiological constraints and the plausible geometric constraints on operators would suggest fewer than the 40-odd operators that we have used [or than the 30-odd used by Doyle (1960) or the 75 used by Bledsoe and Browning (1959) ignoring the fact that they cannot be so easily interpreted neurophysiologically]. For example, straight-line and sharp-curve operators would seem to be more plausible in terms of the ease of connection and the importance of the information to which they respond. A possible operator that might overcome this problem, with which we are now working, is a simple differencing operator that will, by means of several additional layers of operations, first delineate contour and then compute successively higher order differences, and hence straightness, slope and curvature, for the unknown pattern. This operator appears to be equivalent to a simple net of excitatory and inhibitory elements (Uhr and Vossler, 1961*a*).

This, then, suggests that the mapping part of the program would be effected by two layers of parallel basic units in a neuron netlike arrangement. The matching part might similarly be performed by storing the previously mapped lists in a parallel memory and sweeping the input list, now mapped into the same standard format, through these lists. Finally, the amplifiers can be interpreted as threshold values as to when the differences thus computed lead to an output. The specific pattern characteristic amplifier would be an additional single unit layer lying right behind the memory list; the interpretation of the general amplifiers might be made in terms of chemical gradients, but is more obscure.

Thus a suitable parallel computer would perform all of the operations of this program in from three to ten serial steps. This is a somewhat greater depth than those programs, such as Selfridge's (1959) and Rosenblatt's (1958), that attempt to remain true to this aspect of the visual nervous system. But it is well within the limits, and actually closer to the specifications, of that system. It also takes into consideration the very precise (and amazing) point-to-point and nearness relations that are seen in the visual system, both between several spots on the retina or any particular neural layer, and from retina to cortex (Sperry, 1951). It also is using operators that seem more plausible in terms of neural interconnections—again, in the living system, heavily biased toward nearness.

The size of the over-all input matrix has also been chosen with the requirements of pattern perception in mind. Good psychophysical data show clearly that when patterns of the complexity of alphanumeric letters are presented to the human eye, recognition is just as sure and quick no matter how small the retinal cone mosaic, until the pattern subtends a mosaic of about the 20×20 size, at which time recognition begins to fall off, in both speed and accuracy, until a 10×10 mosaic is reached, at which point the pattern cannot be resolved at all. This further suggests something about the size of the basic operator, when we consider that most letters are composed of loops and strokes that are on the order of ½ or ¼ of the whole. For our present purposes, the advantage of the 5 × 5 operator was not only its plausibility but also the fact that it cuts down to a workable size the space within which to generate random operators of the sort we are using when we permute through all possible combinations of the matrix. Again, with the constraint that these random operators be connected, it becomes a more powerful geometry- and topology-sensitive operator, and also a simulation of a more plausible neural net.

Finally, psychophysical evidence also strongly suggests that the resolving power of the human perceptual mechanism is on the order of only two or three bits worth of differentiation as to dimensions of pattern characteristics—things such as length, slope, and curvature (Alluisi, 1957; Miller, 1956*b*; Uhr, 1959*b*). This, again, suggests a 5 × 5 matrix as a minimum matrix that is capable of making these resolutions.

The specifications for and methods used by living systems, and especially the human visual system, suggest certain design possibilities for a pattern-recognition computer; but they certainly do not suggest the only possibilities. Nor should they be slavishly imitated. They should, however, be examined seriously, for the living pattern recognizers are the only successful systems that we know of today. Nor does it seem that the sort of use we have been making of these human specifications will impose any fundamental limitation on a program such as this, one that generates and adjusts its own operators. We have, in fact, already found the program making a different, and, apparently, more powerful, choice of operators than the choice suggested to us by the psychophysiological data and conjectures we have just described. The program's "learning" methods can now depend both on built-in connections (maturation) and on the inputs that need to be learned. The program will develop differently as a function of different input sets. It appears to be capable of extracting and successfully using information from these sets. This would seem to be as completely adaptive—being adaptive to inputs—as a computer or organism can be expected to be.

This sort of design would seem to have some applicability to a variety of more "intelligent" machines. The program replaces the programmer-analyst by a programmed operator that first generates operators that make effective enough use of the unknown input space, and then makes use of feedback as to the success of these new operators in mapping unknown inputs in order to increase their effectiveness. Thus neither programmer nor program needs to know anything specific about the problem ahead of time. The program performs, as part of its natural routine, the data collection, analysis, and inference that is typically left to the programmer. This would be a foolish waste of time for a problem that had already been analyzed. But pattern recognition, and many other problems of machine intelligence, have not been sufficiently analyzed. The different pattern-recognition programs are, themselves, attempts to make this analysis. As long as pattern recognition remains in the experimental stage (as it must do until it is effectively solved), a program of this sort would seem to be the most convenient and flexible format for running what is, in effect, a continuing series of experiments upon whose results continuing modifications of theories are made. This becomes an extremely interesting process for the biologist or psychologist, especially to the extent that the program can be interpreted either physiologically or functionally, or at the least does not violate any known data. For the experimentation and concomitant theory building and modification being undertaken today is rapidly building what appears to us to be the first relatively firm and meaningful theoretical structure—for pattern, or form, perception—for the science of "higher mental processes."

Self-generation of operators, by the various methods employed in this program, may also suggest approaches toward solving a wide variety of pattern-recognition and pattern-extraction problems. Thus there is some hope that relatively powerful operators are being extracted and generated as a result of experience with and feedback from the program's quasi-experimental analysis on the body of data that is available to it—its inputs and the consequences of its actions. Further, the level of power of these operators, and the serial ordering of operators can also be placed under similar control. Thus operators need not be overly simple or random to be machine-chosen; nor preprogrammed to be powerful. Rather, they can arise from the problem, and thus be sensitive to the problem, and to changes in the problem.