Lehigh University
Lehigh Preserve

Theses and Dissertations

1968

The use of heuristic methods in problem-solving machines

Darien A. Gardner
Lehigh University

Follow this and additional works at: https://preserve.lehigh.edu/etd

Part of the Databases and Information Systems Commons

Recommended Citation

Gardner, Darien A., "The use of heuristic methods in problem-solving machines" (1968). *Theses and Dissertations*. 3620.
https://preserve.lehigh.edu/etd/3620

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.

THE USE OF HEURISTIC METHODS IN PROBLEM-SOLVING MACHINES

by
Darien A. Gardner

A Thesis
Presented to the Graduate Faculty
of Lehigh University
in Candidacy for the Degree of
Master of Science
in
Information Sciences

Lehigh University
1968

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

January 13, 1968
(date)

Robert T. Barnes Jr.
Professor in Charge

Donald J. Hillman
Head of the Department

Work on this thesis was supported by a
National Defense Graduate Fellowship.

TABLE OF CONTENTS

Page

Certificate of Approval ii

Acknowledgments iii

Table of Contents iv

Abstract 1

The Use of Heuristic Methods in Problem-Solving Machines 3

I Heuristic versus Algorithmic Programs 3

II Definitions of Heuristic 6

III An Example of a Heuristic Program 10

IV Comments on Induction, Learning, and Pattern Recognition 16

V Tree Representations and Levels of Selectiveness 21

VI Heuristics for Method Selection 31

VII Heuristics for Node Selection 43

VIII The Use of Models 47

IX Conclusion 51

Bibliography 53

Vita 57

ABSTRACT

A heuristic method (or simply a heuristic) is a method used by a problem-solving computer program to help discover the problem's solution by making plausible but fallible guesses as to what is the best thing to do next. Heuristic methods utilize partial or uncertain information available during problem-solving to guide the problem-solving machine. The inferences made by heuristic methods are essentially of an inductive type. Heuristic methods are closely related to learning and pattern recognition. This paper divides heuristic methods into three types: (1) those which select methods, (2) those which select nodes, and (3) those which use models. (In logic, "methods" are rules of inference, "nodes" are logic expressions, and "models" are models of the problem situation.) The problems pertaining to the use of each type of heuristic method during problem-solving are examined.

Viewed dramatically, problem-solving is the battle of selection techniques against a space of possibilities that keeps expanding exponentially.

--Newell, Shaw, and Simon, 1959

I

Heuristic versus Algorithmic Programs

A few years ago heuristic and algorithmic programs appeared much more widely separated in aims and methods than they do today. Early attempts to prove theorems in logic were made with both types of programs. The Logic Theorist (LT) of Newell, Shaw, and Simon (1956) and other heuristic programs patterned after it formed a sharp contrast with the exhaustive theorem-proving programs of Wang (1960a) and Gilmore (1960). LT and other heuristic programs were characterized by the use of list-processing languages, the generation of a large tree of intermediate results in the course of problem-solving, and the use of heuristic selection, based on the information provided by these intermediate results, to guide the problem-solver along a fruitful path leading to a solution of the problem. The authors of LT were plainly more interested in understanding heuristic methods and complex information processing than they were in being able to prove theorems in logic. Wang and Gilmore, on the other hand, worked primarily for mathematical results. They had no need for list-processing languages or elaborate storage of intermediate data structures. By applying some of the advanced metatheorems of logic, they were able to reformulate problems in simpler terms and apply more direct exhaustive procedures to them.

As both heuristic and algorithmic programs became more ambitious, there was a tendency for them to borrow from each other. Logicians have long since proved that the predicate calculus is _undecidable_, that is, there is no systematic or algorithmic procedure which can determine, in every case, whether a given sentence in the predicate calculus is a theorem or not. The authors of algorithmic theorem-provers, venturing to attempt ever more difficult problems in the undecidable domain of the predicate calculus, soon began to think about ways to introduce heuristic selection in order to cut down the fruitless expenditure of large amounts of machine time and effort exploring dead-ends. A step in this direction was Wang's use of several pattern recognition methods in his programs (Wang, 1960b; Wang, 1961). Other examples are the suggestions of Davis (1963) for heuristic elimination rules in algorithmic theorem-provers. On the other side of the ledger, the grandly-named General Problem Solver (GPS) which grew out of LT, is able to solve a variety of different problems, among them the finding of proofs for theorems in logic. Profiting from the experience of the algorithmic theorem-provers, the latest version of GPS (Ernst, 1966) incorporates both the unit preference strategy used by Wos _et al_ (1964) and the resolution principle used by Robinson (1965) without much difficulty into its heuristic framework, thereby much increasing its theorem-proving capacity. On the one hand, although algorithmic problem-

solvers continue to surpass heuristic ones in the area of logic, we can agree with Cooper (1966) that, at a certain level of difficulty, "heuristic methods will be needed in theorem proving." On the other hand, heuristic problem-solvers, taking advantage of available algorithmic procedures, can proceed into areas where no algorithms are known. The heuristic program of Slagle (1961a) to solve analytic integration problems is an example. Although confined to problem-solving in logic, this brief discussion is intended to show that heuristic and algorithmic approaches can usefully be regarded as complementary, rather than antagonistic.

II
Definitions of "Heuristic"

What is meant, exactly, by "heuristic"? The word is very convenient, slippery, and hard to define. Webster's Third New International Dictionary of the English Language gives

> heuristic, adj. -- serving to guide, discover, or reveal.

In artificial intelligence the word has a somewhat more specialized meaning. A sampling from reports on heuristic programs yields the following definitions.

> A heuristic method (or a heuristic, to use the noun form) is a procedure that may lead us by a short cut to the goal we seek or it may lead us down a blind alley.
>
> (Gelernter, 1958, p. 337)

> A process that may solve a given problem, but offers no guarantees of doing so, is called a heuristic for that problem.
>
> (Newell, 1957, p. 114)

> A heuristic method is a provisional and plausible procedure whose purpose is to discover the solution of a particular problem at hand.
>
> (Gelernter, 1959, p. 135)

> 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.
>
> (Slagle, 1961b, p. 192)

> Heuristic methods are rules that, with relation to some specific problem-solving task, are likely to work in a large proportion of cases but are not guaranteed to do so.
>
> (Cooper, 1966, p. 163)

All of these agree in asserting the fallibility of heuristics. A heuristic has the virtue of plausibility. It is intended to be helpful, but its fallible guesses will sometimes lead us down a blind alley. One might also gather the impression that heuristics are *ad hoc*, arbitrarily introduced by the programmer and not good except for the particular problem at hand. In Sections 5-8 of this paper a more general framework for heuristics is outlined which may remove much of their seeming arbitrariness.

A misleading definition which reappears under various guises in the literature is:

> A heuristic is any principle or device that contributes to the reduction in the average search to solution.
>
> (Simon, 1958)

The objection to this definition is in the broad inclusion suggested by the word "any". It is true that heuristics are supposed to reduce the average search in problem-solving activity. But the definition as given includes a whole different class of improvements, namely, improvements in problem formulation or representation. A trivial example of an improvement in representation is the change from Roman numerals to Arabic numerals: I can compute 1776 subtracted from 1968 is 192 much more easily than MDCCLXXVI subtracted from MCMLVIII is CXCII. Non-trivial examples can easily be given. And indeed, major improvements in algorithmic theorem-proving have depended on recasting problems in a more easily proved form. (Compare Ernst, 1966, p. 41: "Perhaps it is best to

consider different formulations of a problem to be different problems.") The reformulation of a problem in simpler terms may reduce the amount of search required without being a heuristic. Returning to our example above, the statement of a numerical problem in Arabic numerals is logically equivalent to its statement in Roman numerals. Translation from either one into the other is not fallible because it works on every occasion. It should be observed, however, that if a number of alternative forms of representation are available to solve a given class of problems, the grounds for selecting one will generally be heuristic, since the representation selected may be infeasible on some of the problems, or another form of representation not considered might have been even better.

Since everyone seems to have his own definition of "heuristic", it is only appropriate in concluding this discussion of definitions to offer one also. For the purposes of this paper a useful definition of "heuristic" is simply "a method of selection." Making a slight generalization on Slagle (above), we have: Given a range of alternatives, a <u>heuristic</u> is a method which helps in discovering a problem's solution by making a plausible but fallible selection of those alternatives which are most promising. This selection may take any one of several forms. It may choose just <u>one</u> of the alternatives as the best. It may <u>order</u> the alternatives, placing the most promising first and the least promising

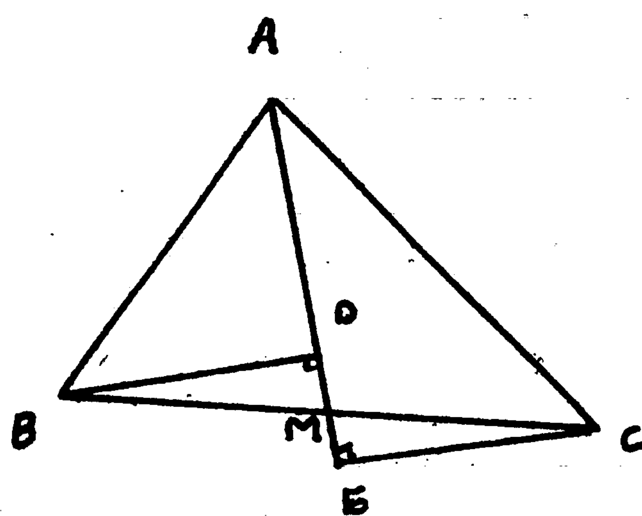
last. Or it may eliminate some of the alternatives entirely, without discriminating among the rest. All three types of selection are very common in heuristic programs. Their function is to guide the problem-solver to make plausible -- and hopefully right -- choices.

III
An Example of a Heuristic Program

Although the Geometry Machine of Gerlernter (1958, 1959, 1960) for finding proofs in plane geometry can be made to serve in most respects as a typical heuristic program, it is distinguished by its heuristic use of a model. The model in this case is a simulation of the ruler-and-compass diagram which would be used by humans in solving such problems. We always draw diagrams in solving plane geometry problems, either on paper or "in the head." The diagram is an interpretation (hence a semantic model) of the formal axiomatization which underlies Euclidean geometry. Humans find Euclidean diagrams intuitive and non-Euclidean ones counter-intuitive, but the machine would derive heuristic guidance from a semantic model in any formal system without regard to its intuitive appeal. To the Geometry Machine non-Euclidean diagrams (also simulated) for proving non-Euclidean theorems would have been just as feasible. Euclidean geometry was chosen for convenience of exposition and because the familiarity of the subject would allow direct comparison with the heuristics used by humans in problem-solving, as well as easy evaluation of the errors and successes of the machine. Although decision algorithms for plane geometry are available, Gelernter deliberately ignores them in order to investigate heuristic processes:

> The object of this research has not been to design 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.
> (Gelernter, 1959)

Like the LT program of Newell, Shaw, and Simon, the Geometry Machine tries to find a proof by "working backwards" from the goal to the premises. An example of a theorem to be proved is: "Two vertices of a triangle are equidistant from the median to the side determined by those vertices." The problem is to show BD equals EC in the diagram shown in Figure 1.



Given BM equals MC
Prove BD equals EC

Figure 1

The program begins by surveying the diagram, listing every segment, every angle, and every triangle. The premises are added to the list of established formulas. The statement to be proved is designated as G_0, which means the zero-order or original goal. Then the axioms and theorems which might be applied to yield G_0 are selected. Any condition or

combination of conditions which, in conjunction with a theorem or axiom will yield G_o in one step (by modus ponens) constitutes a first-order goal (G_1). As the process continues a problem-solving "tree" is generated, with each N-order goal generating a number of (N+1)-order goals. The program is recursive: all the techniques and heuristics available to the program for attacking G_o can be applied to successive goals because the programs uses itself as a subroutine. Since each subgoal (in this formulation) implies the goal which generated it, if at any point a subgoal can be immediately inferred from the established formulas, the theorem is proved.

The major heuristic in the system is the diagram, simulated in such a way as to give exactly the same information to the machine as a pictorial diagram would give to a human. "Working backwards" is the strategy which allows effective use of the diagram. Working backwards has the advantage that every sequence thus generated terminates in the desired G_o. It has the disadvantage that the great majority of these sequences go off blindly into space, having no validity either in the diagram or the theorems and axioms. When every new subgoal is checked for validity in the diagram, the number of non-redundant branches from a generating goal is reduced from <u>several hundred</u> to an average of about five. Without the selectiveness of the diagram there would not even be space to store the first-order subgoals (Gelernter <u>et al</u>,

1960). Another point which emphasizes the effectiveness of the semantic model is the ability of the Geometry Machine to explore to a depth of twelve levels or more. (The depth attained by LT, though not strictly comparable, was three or at most four levels.) It is interesting to note that the diagram, by virtue of its incompleteness (see Minsky, 1956, Chapter II), errs on the side of being too lax, as it should, rather than on the side of being too strict. Accepting one more subgoal that leads nowhere is not too serious; it just means a little more work for the machine. Rejecting a true statement, on the other hand, may be quite serious, since it carries with it a good chance of causing the machine to miss a proof. The diagram in the Geometry Machine makes both types of error. Yet even a much less exact, cruder model would be much better than no model at all.

Several experiments were run to compare the performance of the machine on various problems with, and without, the addition of minor heuristics. Overshadowed by the use of the diagram, they are called "minor" here although they would occupy the foreground in most heuristic programs. Following the divisions adopted in Section VI and Section VII of this paper, there were two types of minor heuristics:

<u>method selection</u> - The number of transformations WAS REDUCED by selecting only certain theorems to be applied in generating subgoals, depending on the type of the generating subgoal.

node selection - Instead of trying to solve the subgoals on the stored subgoal list in the haphazard order in which they were generated, the Geometry Machine acquires a "sense of direction" by:

(1)attempting first those subgoals which can usually be established in just one step, e.g. the equality of vertical angles;

(2) attempting the subgoals which are "closest" (in a well-defined sense) to the premises before others.

The heuristics for node selection resulted in an ordering, rather than an elimination, of subgoals. With the help of the minor heuristics the machine performed substantially better than without them. For example, in the problem illustrated in Figure 1 the machine found a proof in about eight minutes without minor heuristics. A proof was forthcoming in about one minute with the expanded set of heuristics. In addition, the difference between the two proofs is quite striking: the first is long, irrelevant, and seems to get nowhere, whereas the second is short and to the point. Sixty-one subgoals on twelve levels were generated in the first case, as compared with twenty-one subgoals on three levels in the second (Gelernter, 1959).

The Geometry Machine illustrates the basic character-

istics of heuristic programs in a relatively pure form. A proof, when found, is definite and free from the optimization-of-solution problems usual in practical applications (Tonge, 1961; Kuehn and Hamburger, 1963; Karg and Thompson, 1964) of heuristic programs. Nor does the Geometry Machine have to cope with the complexities introduced by a hostile environment as do the game-playing machines (Newell *et al*, 1958a; Samuel, 1959). Basic characteristics of heuristic programs may be summarized as follows (compare Tonge, 1961):

1. *Subdivision of Problems* -- application of methods to generate subgoals or otherwise subdivide the problem into (easier) parts. Methods may range from using modus ponens to generation of new models.
2. *Use of Heuristics* -- exploitation of partial information in the course of problem-solving to select the best alternatives. This is where the opportunity to use learning and pattern recognition arises.
3. *Recursiveness* -- bringing to bear on a subproblem all the techniques and methods that were available for the original problem.
4. *Fallibility* -- ability to bypass a solution through wrong "guessing." This is possible because only a partial search is made, not an exhaustive one.

IV
Comments on Induction, Learning, and Pattern Recognition

A common sense example of induction is the following. A small college has 700 students with 185 students in the freshman class this year. Next year the college would like to increase the size of the freshman class to about 200. Of course, more than 200 applicants must be accepted because many will prefer other colleges and others will not come for a variety of other reasons. Then how many should be accepted? The college may reason inductively that in past years close to two-thirds of the students originally accepted came, and therefore this year 300 students should be sent letters of acceptance. This is based on the plausible assumption that in a highly similar situation next year the proportion (two-thirds) of "successes" will be similar. There is no guarantee that the guess will be correct, and on many occasions colleges have received fewer than they expected, or have been obliged to turn intended single rooms into doubles.

A straightforward example of a heuristic based on the same principle is Slagle's (1961) measure of the _depth_ of an expression to be integrated, that is, the maximum level of function composition contained in it. (Depth is only one of the eleven features his program uses to characterize integrands.) Thus

x is of depth 0,

x^2 is of depth 1,

e^{x^2} is of depth 2,

xe^{x^2} is of depth 3.

It is evident that the depth of an expression gives a crude measure of its difficulty. Slagle considered several other measures for estimating how much work would be required to integrate an expression and settled on this simple one. The importance of this measure is not only the proportion of successful integrations (though the proportion would be higher for the easier problems) but also the relative cost or effort needed. Slagle's machine (called SAINT for Symbolic Automatic INTegrator) orders the subgoals according to their depth and attempts first the ones that appear easiest. The Geometry Machine, as mentioned earlier, uses a two-part division on the same principle when it attempts first all those goals which are usually established in one step. That the depth of an expression is not always a good index of its difficulty is shown by the fact that $\int e^{x^2}dx$ with a depth of 2 in the integrand cannot be integrated in elementary form, while $\int xe^{x^2}dx$ with a depth of 3 is a relatively easy problem. It should be also remarked that attempting the easier goals first may not be a sufficient guide by itself. In complex problems where a great many goals have been generated, some measure of centrality of importance to the main goal may be needed to prevent the problem-solver

from spending its time on the easiest subproblems while ignoring the real difficulties.

Slagle's SAINT program is not a learning machine. If the same problem is given to it a second time, it will do the same operations, make the same mistakes, and come to the same conclusion. Nor is the storing of proved theorems by theorem-provers what we mean by learning, since most of the heuristic theorem-provers, if the theorem were erased from memory and given again to be proved, would go through exactly the same steps as before. On the other hand, if the heuristics are improved or modified in the course of problem-solving, then the future performance of the machine will show differences, without the external intervention of the programmer. The point is that we would like our machines to adapt themselves to the type of problems they are called upon to solve, that is, to learn. Learning requires generalization on past experience, which entails inductive inference.

If the learning process is accepted as inductive in nature, it follows that pattern recognition, since it involves learning, is also inductive. To see the relation of learning and pattern recognition as they might be useful in a problem-solving machine, let us consider how a machine like SAINT could have learned, for itself, a better order in which to attempt subgoals. Assume that the method for computing "depth" as a feature of the integrand is already given. Then

SAINT would be able to correlate this with "success" and find for itself the optimum order (for this feature) in which goals should be attempted to reduce effort. If there were a number of features, the familiar techniques of optimization and correlation could be applied. The problem-solver then would be learning and adapting in a way which would affect its performance on all future problems. Now let us assume that depth, as a feature, is <u>not</u> given beforehand, and further that the problem-solver is capable of originating it. Then it has the basic requirement for pattern recognition, namely, originating its own features (Selfridge and Neisser, 1960; Uhr and Vossler, 1961). It is true that a learning machine also, in a sense, recognizes patterns. The patterns it can recognize are those for which its pre-given features are suited and usually those which its programmer foresaw. A pattern-recognizer, on the other hand, is much less limited. By devising new features it may <u>discover</u> patterns which the programmer never thought of and for which he has no intuitive counterpart.

The possible uses of learning and pattern recognition to a heuristic problem-solver are several. By learning methods alone the problem-solver may put to work a feature that has been pre-computed for it, using it to help select among alternatives (as "depth" does in the example above). Learning and pattern recognition together have great potential, including (1) originating new, useful features and

thus permitting heuristic selection in ways unthought of by the programmer; (2) discovering patterns in the proof-search procedure for the class of problems given to it and checking for various patterns before falling back on lower-level search; and (3) discovering patterns which turn out to be useful models or representations.

If learning and pattern recognition have so much potential (it will be asked), why don't we use them? The answer is that some steps in that direction are being made. The GPS program (Ernst, 1966) uses learning, as does the early heuristic checker-playing machine of Samuel (1959). (The checker-player once won against a checkermaster, though on a rematch there were five draws and one win for the man.) There have been several heuristic problem-solvers which discover plans and models of various types (Newell et al, 1959; Tonge, 1961; Berlekamp, 1963; Travis, 1964). There have been machines which discover new transformations (Evans, 1964). But the movement in this direction has been much less than we might hope, for a simple reason: pattern recognition is one of the things which humans do very well and which machines do very badly. Therefore the programmer of a heuristic problem-solver usually does the learning and pattern recognition necessary to discover heuristics, and the machine accepts them passively.

V
Tree Representations and Levels of Selectiveness

Finding a solution to a problem may be regarded as a problem of search: search through a set of possible solutions to find one which is acceptable. For example, the particular proof that a theorem-proving machine is looking for belongs to the set of all possible proofs. The shortest proofs are of most interest because the difficulty of a theorem generally increases exponentially with the length of the proof. The theorem-prover wants to find a sequence of statements, each one following from the previous ones by a valid rule of inference, with the last one being a statement of the theorem. A convenient representation for this and many other problems is the problem-solving tree shown below (Figure 2).

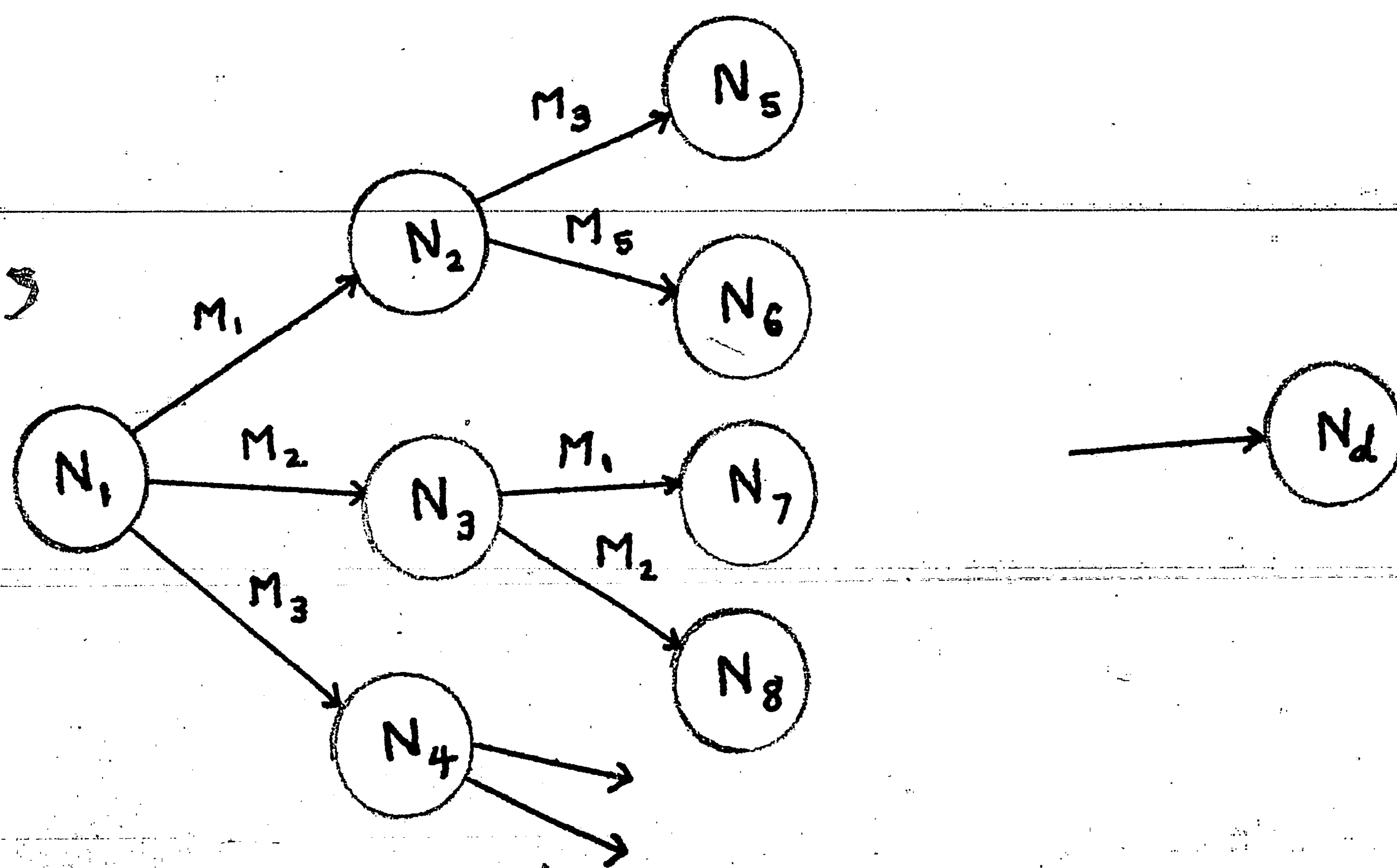


Figure 2.
A tree representation

Ordinarily node N_1 in the figure would represent the beginning situation and node N_d would represent the desired situation. "Working backwards" in theorem-proving means that N_1 is the theorem to be proved and N_d is the conjunction of the established formulas. Then the methods (rules of inference) M_1, M_2, M_3, . . . work "in reverse," so that if any of the later nodes can be shown to be true, all the previous nodes in the sequence which led up to it are automatically valid. As remarked earlier, this formulation has the advantage that every sequence will terminate in the theorem N_1, and the disadvantage that the later nodes may very well be false. In the Geometry Machine this is a net advantage, since most of the false nodes can be eliminated by comparison with the diagram.

As Minsky (1961a) observes: "Almost any [well-defined] problem can be converted into a problem of finding a chain between two terminal expressions in some formal system." In game situations the nodes N_1, N_2, N_3, . . . represent board positions and the methods connecting one node with another are the legal moves that may be made in those positions. Once the tree has branched, the branches cannot come together again, although some nodes may be identical with others. In chess and checkers the number of different nodes is finite, so that it is theoretically possible to discover the best move(s) in any given position by computing every possible continuation out to the bitter end. In

practice, such a systematic procedure would take an unthinkably long time. For additional steps explored the amount of time increases exponentially, given by

$$\text{Total time} = \text{Unit time} \times B^D$$

where B is the amount of branching per node and D is the depth of exploration. Thus checkers is "solvable" in this manner, but to explore the approximately 10^{40} continuations, at three move choices per millimicrosecond, would take 10^{21} centuries (Samuel, 1959). Chess, with 10^{120} continuations, would take 10^{80} times as long. Interesting problems in general cannot be solved by simple enumeration of possibilities because of the prohibitive size of their search trees.

At any given time the growth reached by the tree is a complete representation of the results reached so far in the problem-solving process. The problem-solver would maintain (1) a record of the relations between the nodes and of the methods by which they had been generated, that is, the equivalent of the graph in Figure 2; (2) a record of descriptions specifying what each node represents, for example, descriptions of chess situations represented by nodes; (3) attached to each node, a record of the features which have been computed for that node; and (4) a record of the part of the tree which is currently under attack. Given a node N_p, the set of computed features $F_1(N_p)$, $F_2(N_p)$, $F_3(N_p)$, . . . might be either numbers or linguistic

expressions. Since features are supposed to help in problem-solving and suppress irrelevant information, they should be small in size -- certainly smaller than the full description of the node N_p. The problem-solver should err on the side of having available more features than necessary rather than not enough. If computing correlations between features and methods becomes too time-consuming, the marginal features can be discarded, or some can be temporarily suspended. Samuel's checker-player uses 38 features for checker positions, of which 22 are held in reserve and rotated with the 16 used during play.

In the course of working on a problem the problem-solver tends to find the overall situation more and more complicated as the problem tree grows and partial results accumulate. In this kind of situation it becomes apparent that there is an administrative problem in deciding at what level selectivity should be applied. Three levels from most specific to most general may be distinguished: (1) selection of methods based on the features of particular nodes; (2) selection of nodes based on a review of the problem tree; and (3) construction of plans or models of the problem situation.

Selection of Methods. Let us suppose we are in the midst of a problem, a number of intermediate nodes have been generated, and the next node N_p to be worked on has already been chosen. Of methods M_1, M_2, M_3, . . . which one shall we apply to N_p? In general that will depend on

the features of N_p. If the machine has encountered nodes previously which are "similar to" or "have featues in common with" N_p, and if the machine has correlated these features and the success of its methods, then it will try the methods which were most successful before. Features are "good" solely relative to the methods used by the machine. Since new situations cannot be expected to be exactly the same as old ones, some such generalization technique is necessary. The basic pattern which underlies this type of learning process is:

> N_p is similar to N_q.
>
> Method M_k worked on N_q.
>
> Therefore M_k has a better chance of working on N_p than it would have a priori.

As a result of this process certain methods are estimated to be more promising than others. The problem-solver conserves effort by never trying to apply those methods which seem unpromising, and the goals which they might have generated never appear on the problem tree. The recursive character of the selection is clear: only (n + 1)-order nodes produced by methods selected for n-order nodes can be focal points for a new method selection process. Even a small improvement in such a recursive process has a great advantage over uniform improvements like increased speed. If we compare increased speed and increased selectiveness as sources of improvement in (say) a chess-playing

program, we observe that doubled speed would take its effect once for the whole game, while doubled selectiveness in methods would halve the number of branches at every step. The chess-playing program of Bernstein et al (1958) obtains an even greater reduction. By using plausible move generators it reduces the number of moves considered from the thirty or so moves possible in typical chess positions to only seven. Thus the growth of the search tree is cut down drastically. This is a step in the direction of human practice, since good human players, who play considerably better than the best chess-playing machines to date, consider many fewer than seven moves at each step.

Selection of Nodes. What happens after a method is chosen and a new node is generated? If full attention were given to each node, as generated, the problem solver might never get back to the alternate branches of the tree. Two fixed orderings which have been used to decide which node to take next are (1) depth-first, and (2) breadth first. Samuel's checker-player uses a depth-first ordering, as in Figure 3 on the following page.

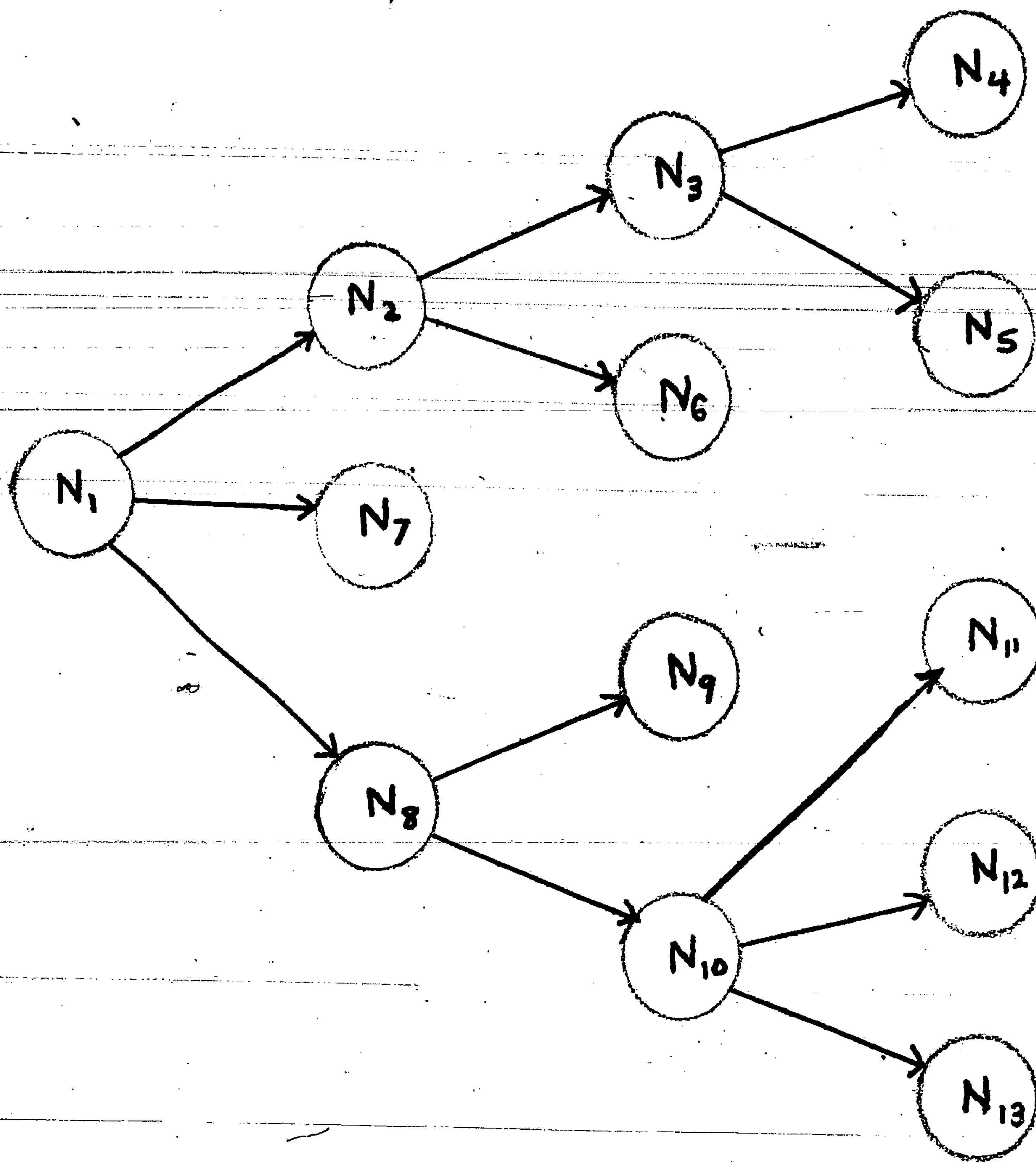


Figure 3.
Nodes ordered depth first

For this type of ordering a cut-off point at a certain depth must be given, either predetermined, or partially determined by the features of the nodes as in Samuel's program. On the other hand, a breadth-first ordering (for example in LT) will generate all the nodes on one level before it proceeds to the next. The numbering of the nodes in Figure 2 above shows this type of ordering.

Rather than being bound to follow a special order, any powerful heuristic program will have the ability to select what node to work on next. Thus the problem-solver might

decide to skip to an entirely different part of the tree. It is typical of human problem-solving, for example, that success in one subproblem of a larger problem may greatly affect the prospects for solution of another subproblem. Partial failure, though usually less dramatic, will affect the evaluation of different parts of the problem. For the machine, also, as the problem tree becomes large and successes or failures are encountered it becomes increasingly important to be able to review the entire tree. Selection of the next node to work on will depend in part on (1) estimates of the difficulty of the nodes, considered as individual problems, and (2) the centrality of nodes for establishing the main goal. These estimates may require new features to be added to those already used for method selection. Estimates of difficulty are important because the problem-solver wishes to find an easy path from N_1 to N_d. The easiest nodes may be attempted first, and nodes which appear especially difficult, or appear more difficult than the higher-level nodes which they are supposed to help solve, may be avoided. Estimates of centrality require using difficulty estimates between individual nodes to find minimally difficult paths between N_1 and N_d. Nodes which are central -- that is, on one of these minimal paths -- will be favored.

It should be pointed out that there are a number of difficult questions connected with reviewing the problem tree. The very rapid growth of the tree makes it increasingly

difficult, nor does a full review seem justified at every step. Compromise methods for reviewing just a part of the tree are attractive (one such method is discussed by Minsky, 1961a). Another difficulty concerns what to do when a partial success is achieved. In a complex problem the solution of an important subproblem is not independent of the circumstances of its creation. Methods needed to evaluate the resulting partially modified state of affairs may be quite sophisticated (Newell, 1962a).

Plans and Models. Constructing plans or models for a problem permits heuristic selection on an even higher level. After review of the entire problem tree, pattern recognition would appear to be a natural means for helping to construct a plan or model for the problem, but no heuristic program to date has used pattern recognition in this way. Two main types of models may be mentioned: the semantic model and the analogous model. The first type, the semantic model, has already been discussed (Section III). The second type is the analogous (or simplified) model which Minsky (1961a) describes as follows:

> Perhaps the most straightforward concept of planning is that of using a simplified model of the problem situation. Suppose that there is available, for a given problem, some other problem of "essentially the same character" but with less detail and complexity. Then we could proceed first to solve the simpler problem. Suppose, also, that this is done using a second set of methods, which are also simpler, but in some correspondence with those of the original. The solution to the simpler

> problem can then be used as a "plan" for the harder one. Perhaps each step will have to be expanded in detail. But multiple searches will add, not multiply, in the total search time. The situation would be ideal if the model were, mathematically, a homomorphism of the original. But even without such perfection the model solution should be a valuable guide [his italics].

In making this summary review of different levels at which heuristics may be applied, I am conscious of having glossed over many (mostly unsolved) problems which are not, fortunately, too closely related to my main topic. Some of these problems are problems of effort assignment, such as: How much of the total effort should be devoted to exploring the problem tree? How much to discovering new features? How much to reviewing and comparing nodes? How much to trying to discover good models? And so on. The administrative problems appear formidable. I am content in having presented a loose framework and at least a partial justification for the division of heuristics into the three types of the next three sections.

VI
Heuristics for Method Selection

The examples given below are intended to be representative of the different types of heuristics for method selection found in heuristic programs. The papers referred to in this and the next two sections are primarily Baylor and Simon, 1966; Ernst, 1966; Evans, 1964; Gelernter, 1958, 1959, 1960; Kuehn and Hamburger, 1963; Karg and Thompson, 1964; Newell *et al*, 1956, 1957, 1958a, 1959, 1960; Samuel, 1959; Slagle, 1961a, 1961b; Simon, 1963; Simon and Simon, 1962; and Tonge, 1960, 1961, 1963. In cases of choice it has been convenient to give as examples the heuristics which have been more fully documented in preference to others less fully described in the literature.

The Similarity Test in LT. The LT program of Newell, Shaw, and Simon for proving theorems in logic is based on four methods. The first of these, substitution, tries to prove a theorem by a series of substitutions in the axioms, or in previously proved theorems. All the axioms and previous theorems are considered, but only those which pass the similarity test will be submitted to the full matching algorithm. The similarity test is a screen or filter which prevents LT from wasting time trying to apply methods which are inapplicable. If substitution alone fails, LT generates subgoals by use of three other methods:

detachment, forward chaining, and backward chaining. In detachment, if T is the theorem to be proved, the similarity test, the substitution method, and the matching algorithm are used in order to find appropriate axioms or theorems which can be stated in the form "A implies T." If such an axiom or theorem can be found, A is a new subgoal. In forward chaining, if T is of the form "A implies B" and "A implies C" can be found, then "C implies B" is a new subgoal. In backward chaining, if T is of the form "A implies B" and "C implies B" can be found, then "A implies C" is a new subgoal. The similarity test can apply to wholes or to parts. Thus before trying detachment LT compares the whole of the theorem to be proved with the left hand part (antecedent) of the axiom or theorem to be used. Before trying backward chaining it compares the right hand part (consequent) of the theorem to be proved with the right hand part of the axiom or theorem to be used.

Applying the similarity test depends on certain computed descriptions (features) of logic expressions. The theorem "not-p ⟶ (q or not-p)" may be written in tree form as

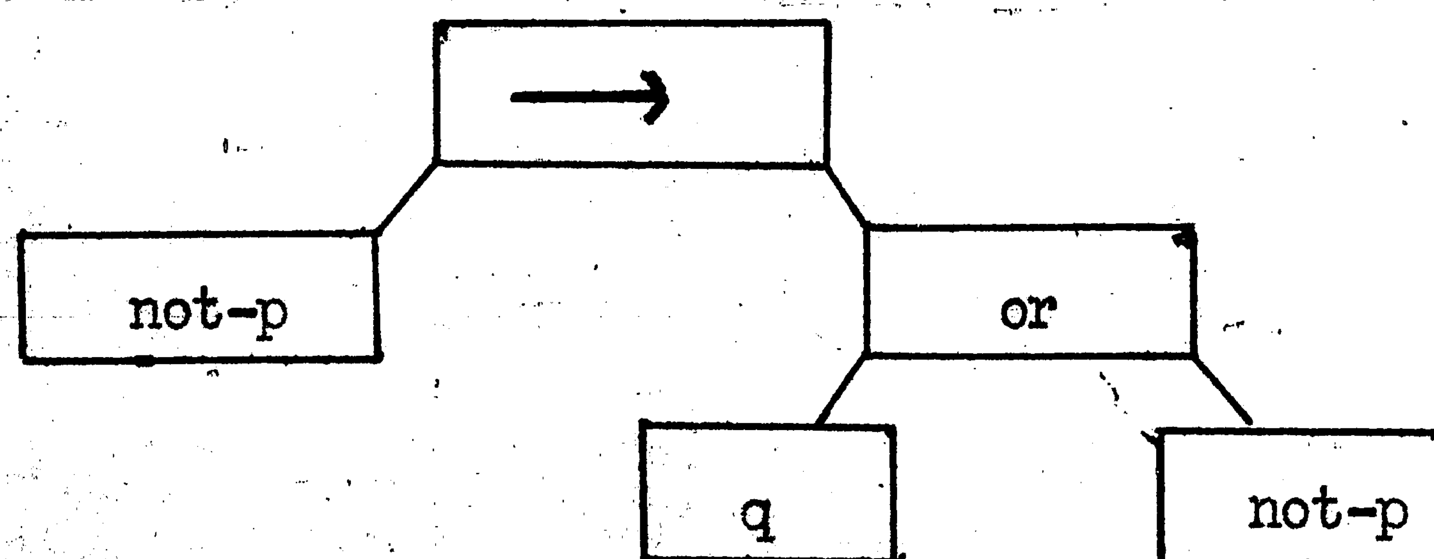


Figure 4.
A logic expression

The computed description refers to (1) the maximum number of levels from the main connective to anyvariable; (2) the number of distinct variables; and (3) the number of variable places. The left hand part "not-p" has the description (1, 1, 1), the right hand part "q or not-p" has the description (2, 2, 2), and the whole expression has the description (3, 2, 3). Any other logic expression with the same descriptions is "similar" to this theorem. Since the axioms and theorems are reviewed again and again during problem-solving, LT computes all their descriptions at the start. The number of candidates for matching rejected by the similarity test runs between 75% and 95%. For example, out of 11,298 candidates for the substitution method, only 993 passed the similarity test, and of these, only 37 matched (Newell _et al_, 1957). A more general notion which permits a somewhat more flexible application of the similarity test is described by Newell _et al_ (1956).

Occasionally the similarity test makes mistakes. For example, the theorem

$$p \longrightarrow (p \text{ or } p)$$

can be proved simply by substitution of p for q in the axiom

$$p \longrightarrow (q \text{ or } p).$$

Therefore this axiom is highly appropriate, but the two expressions are called dissimilar by the similarity test because the first contains only p on the right while the second contains p and q. LT discovers a proof after chaining

forward, but at the cost of about five times as much effort as without the similarity test.

Reduction of "Differences" in GPS. The experience of Newell, Shaw, and Simon with LT led them to create the more sophisticated General Problem Solver. As its name implies, GPS is aimed at achieving a greater degree of generality. Using the problem tree as a representation, it seeks to obtain problem-solving heuristics independent of the subject matter, that is, to use the same heuristics to deal with different subjects. At first, problems of representation limited GPS to only a few tasks, but ultimately GPS's potential for handling a wide variety of tasks was realized (Ernst, 1966), including problems in the first-order predicate calculus, analytic integration, and parsing sentences.

GPS's procedure for selecting the next method to apply to a node is by obtaining a difference between the present node and the desired node. The authors of GPS reason, quite logically, that if the present node is not the desired one, a difference will be detectable between them. The difference is then made to help choose an appropriate method. In effect, the desired node is being used to generate a feature for the present node. For different subject matters different differences will be found, and different methods will be appropriate to "reduce" these differences.

To illustrate the use of differences, and contrast

with the approach used by LT, an example from Newell et al (1959) may be given. The problem is to show that from

r and (not-p ⟶ q)

we can deduce

(q or p) and r.

GPS uses a matching algorithm to find a difference and sees that "r" occurs on different sides of the main connective "and". Therefore it looks in its difference-method table (or Table of Connections) under "change position" and finds that the axiom "(A and B) is equivalent to (B and A)" is appropriate. It obtains

(not-p ⟶ q) and r.

GPS again asks what is the difference between this new node and the desired node. This time the difference is on a lower level: the connectives "or" and "implies" in the left subexpressions are different. Looking in the Table of Connectives, GPS finds "(not-A implies B) is equivalent to (A or B)". It obtains

(p or q) and r.

Applying "(A and B) is equivalent to (B and A)" once again, GPS finds the desired goal

(q or p) and r.

As might be expected, the effectiveness of differences depends on how specific the Table of Connections can be made. The connections for the Tower of Hanoi puzzle are ideal: GPS never makes a mistake. On the other hand, Ernst (1966, p. 210) acknowledges that "For many tasks a good set of differences and a good DIFF-ORDERING are difficult to obtain."

GPS does not give up easily in trying to apply its methods. If a difference X_d is obtained for a node X and a method X_m is indicated by the Table of Connections for "reducing" X_d, but X_m is inapplicable because of the form of X, then GPS will set up a new goal Y to cast X in a form acceptable to X_m. Thus it obtains a new difference Y_d to find a Y_m. If such a Y_m is found, Y_m and X_m will be applied successively to X to produce a new node. GPS is unusually persistant among mechanical problem-solvers.

Plausible Move Generators in Chess. The plausible move generators of Bernstein (1958) for reducing the number of moves considered in chess have already been mentioned in Section V. The same idea was implemented by Newell, Shaw, and Simon in a more sophisticated manner using the list-processing language IPL-IV. The selection exercised by plausible move generators is a kind of elimination by silence: any move not proposed by a generator is automatically eliminated from consideration by the main program. Each move generator corresponds to some goal in the chess situation: King Safety, Material Balance, Center Control, Development, King-Side Attack, and so on, and each one proposes moves to promote a different goal. Thus only the Material Balance Generator will propose moving out of danger a piece under attack, and only the Center Control Generator will propose P - Q4 as a good move in the opening. Each plausible move generator works independently and would

function just as well if the others were not present.

The use of plausible move generators eliminates over three-quarters of the possible moves in typical chess positions. Bernstein's chess player, since it looks four steps (two moves) ahead, obtains a reduction of about 300 to 1 in the number of continuations which it must consider. In the NSS chess program a preliminary analysis is made at the beginning of each move to select the goals (and generators) which are appropriate to it. Then each generator selects moves. An additional feature in the NSS program is that the moves for the present position are proposed by the main generators, but the continuations are analyzed with the help of a second set of generators called the analysis generators. The continuations are analyzed until they are "dead" _with respect to the goals of the generators_, for example, a position is "dead" with respect to Material Balance if no material is in danger of being won or lost next move. Subgoals can be set up with respect to each of the goals. If the chess program cannot play P - K4 because it would lose the exchange on that square, it will set up the subgoal of first bringing another man to bear on its K4.

It is interesting to note that the "heuristic transformations" of Slagle (1961b) have the character of "plausible move generators." The "heuristic transformations" for integration are defined by exclusion from the "standard forms" (transformations which always work) and the "algorithm-

like transformations" (which almost always work). Slagle says of them:

> 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 . . . . The ten types of heuristic transformation (Slagle, 1961a) used by SAINT are designed to suggest plausible transformations of the integrand, substitutions, and attempts using the method of integration by parts.

Thus out of the whole set of possible transformations and substitutions, which would be very large, only a few plausible ones are generated. We may conclude that elimination by not-being-proposed is a very common heuristic, though often not explicitly mentioned.

Discussion. Each of the three types of heuristic described above appears quite different from the others. LT screens potential methods (applications of particular axioms or theorems) by an abstract similarity test comparing the present node with the axiom or theorem to be applied. The similarity test is a (perhaps more sophisticated) example of all the heuristics which eliminate methods by a test based only on the present node (Gelernter, 1959; Amarel, 1962; Wos et al, 1964). GPS is unusual for selecting methods on the basis of two nodes, the present node and the desired node. But the Table of Connections may be regarded as a simplified form of the correlation of features and methods suggested in Section V. The goals which motivate the plausible move generators in chess

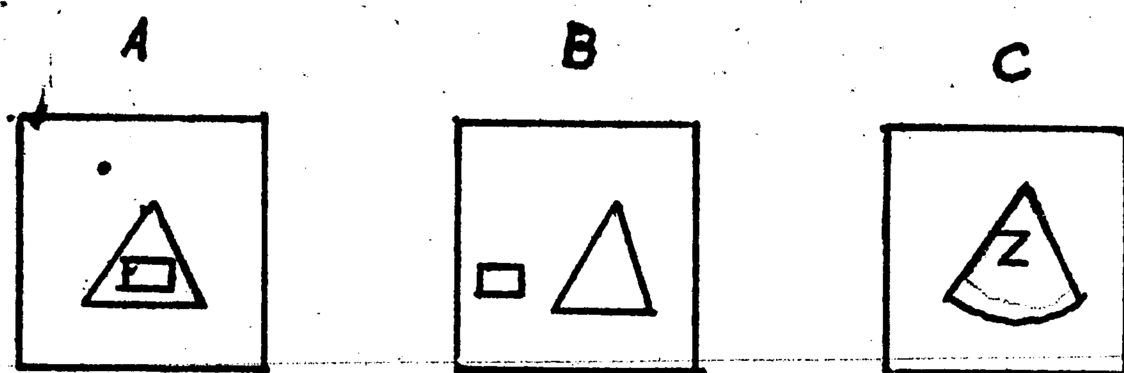
operate on a higher level than either of the two preceding approaches. Goals (such as Center Control) may have no obvious connection with the ultimate goal -- checkmating the opponent's king. They embody <u>features of the game</u>, rather than features of particular positions, and are based on the playing experience of hundreds of players.

In one sense the three examples are similar because they all depend on features. In LT a feature of the present node accepts or rejects candidates. In GPS a feature obtained from two nodes chooses likely candidates. In the chess playing programs each move generator uses a feature to accept or reject candidate moves. It will be noticed, however, that the three operate on successively higher levels of complexity. The similarity test in LT is the simplest. There is no obvious way for it to be improved or for it to improve itself by learning. Then there are the node-differences in GPS, in which learning would be an obvious and natural next step. Actually, GPS is able to learn the Table of Connections in some simple tasks, and is therefore kin to other programs which use the past success of their methods as guides to future application. The most complex examples are the move generators in the chess-playing programs. The tests which they apply to candidate moves do not depend directly on the present or desired nodes but on abstract goals set up by the programmers. These goals really do have importance in chess (otherwise

the machines and humans using them would play very badly indeed) and depend on pattern recognition as applied to whole games. Using pattern recognition to improve these goals (or generate new ones) would clearly be a difficult task. FOR A MACHINE.

Generation of New Methods by ANALOGY. It is important for a powerful heuristic problem-solver to be able to originate new methods, just as it is important for it to be able to originate new features. In some areas, such as chess, the permissible moves are fixed, and combining a sequence of moves into a larger transformation (method) is difficult because of the continual hostile intervention of the opponent. In other areas the discovery of methods which take larger steps or are more desirable in some other way are by no means impossible. The resolution principle of Robinson (1965), which is really a new rule of inference, is a particularly successful example of such a method.

A step in the direction of finding new methods is taken by a heuristic problem-solver called ANALOGY (Evans, 1964). ANALOGY tries to choose the correct answer to geometric-analogy questions of the sort used on intelligence tests. The questions are of the form "A is to B as C is to ? ." In the example given on the following page (Figure 5) ANALOGY agrees with the reader in choosing answer 2.



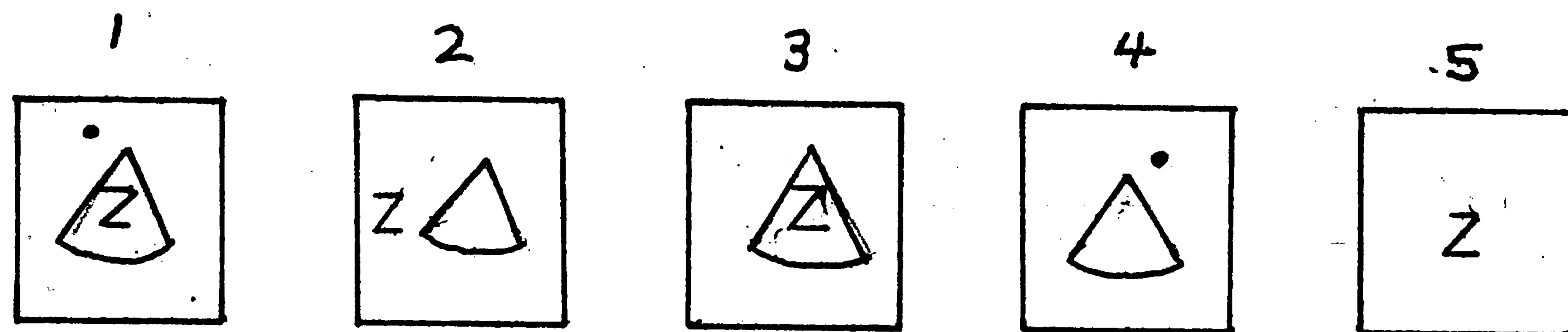


Figure 5.
A geometric-analogy problem

ANALOGY begins by recording a full description of each of the ten small figures, including the spatial relations of "dot," "triangle," "rectangle," and so on. Then it is not too difficult to discover a transformation which carries the description of A into the description of B. This transformation A-B is made as complete and specific as possible. Since the questions are of the form "A is to B as C is to _?_," ANALOGY wants to find a new transformation which will not only carry A into B but also C into one of the answer figures. The original transformation A-B is much too specific to do this, For example, it will make specific mention of "rectangles," and therefore cannot choose one of the answer figures because neither figure C nor any

of the answer figures contains a rectangle. Transformation A-B must be generalized by replacing the description for "rectangle" by a more generalized description. By several generalizations of this kind it is possible to generate a number of new transformations which carry figure A into figure B and figure C into ___ (each one of the answer figures). These new transformations preserve information of differing amounts and kinds from the original transformation. Which one should be selected? The answer of ANALOGY is to choose as "best" or "strongest" the one that says the most or is the least alteration in the transformation A-B and that still maps C onto precisely one answer figure. (If there is a tie, ANALOGY goes on to try a different procedure.)

The importance of this program is that it <u>suggests</u> new methods (transformations) which are not too different from old ones and which are known to be successful. The methods generated in this way are "generalized" and lacking in detail, but details can be filled in depending on the specific problem at hand. (When we set out to apply a method, we rarely know <u>exactly</u> how it will be applied.) Most "suggested" methods will be useless, and will be discarded, but a few may be found useful.

VII
Heuristics for Node Selection

In the earlier parts of this paper the ordering of nodes for selection has already been discussed at various points. Specifically, two examples of fixed orderings (depth-first and breadth first) were illustrated in Section V. Gelernter's use of closest-to-premises and easiest-goals-first criteria were mentioned in connection with the Geometry Machine, as well as Slagle's application of "depth" as a feature for deciding which nodes should be attacked next by SAINT. Section V also discusses some of the problems which can arise in node selection in complex problems.

Further examples of different types of node selection may be given here relatively briefly.

Termination of Search at "Dead" Positions. The checker-player of Samuel (1959) looks ahead either a certain arbitrary number of moves or until it reaches a relatively stable "dead" position. The idea of a "dead" position derives from Turing (Bowden, 1953), who argued that it made no sense to count material on the board until all the exchanges which were occuring had been carried out. Increasing the look-ahead distance for continuations which are "live" with exchanges has the additional advantage of better surveillance of those paths which have better opportunities for losing or gaining an advantage. If there are no special board

conditions, the checker-player looks ahead three steps. At step three the program will continue to look ahead if (1) the next step is a jump, or if (2) an exchange offer is possible, or if (3) the last step was a jump. At step four the program will continue to look ahead under conditions (1) or (2). At steps five through ten the program will continue to look ahead only under condition (1). The look-ahead will terminate at step eleven if one side is ahead by two kings and at step twenty regardless of all conditions. Looking ahead so many steps is only possible, of course, because in checkers branching is highly restricted during exchanges. Here a node's being "dead" or "live" determines whether any further branches will be generated from that node.

Rejecting Harder Subnodes in GPS. GPS has a special criterion for rejecting nodes which derives plausibly from its use of differences to select methods. Underlying the computation of a difference between the present node X and desired node X_d is the idea that the differences between successive nodes and X_d should become smaller and finally vanish. Thus GPS is given (or may learn for itself) a ranking in order of difficulty of the various differences. In logic, for example, a difference of connective is more difficult than a difference of position. Now if it happens that the difference between X_d and a subnode of X is more difficult than a difference between X_d and X, it is plausible

to suppose that the branch taken by the subnode is moving away from the desired node X_d rather than toward it. If this happens, GPS will reject the subnode.

A Special Heuristic. There are many special heuristics which are applicable only in very particular situations. An example is the number-of-branches feature used by the MATER program of Baylor and Simon (1966) for administering checkmate in chess. In exploring the possibilities in a mating problem, MATER gives highest priority to positions which leave the opponent with the fewest replies. Positions in which the opponent's King is in check but can be defended by more than four different moves are discarded entirely. The rationale behind this approach is that positions which allow the opponent much mobility are not likely to lead to checkmate.

Another Special Heuristic. A special heuristic used by the geometric-analogy machine of Evans (1964) is the rejection of answer figures for which there is an incorrect number of parts added, removed, and matched. In the problem illustrated in Figure 5, for example, the original transformation A-B requires that one subfigure ("dot") be removed. As a result answer 1 with three subfigures and answer 5 with one subfigure are rejected. (The situation is more complicated if figure C has a different number of subfigures from A, or if it contains no corresponding "dot".) The heuristic which rejects answer 1 and answer 5 is based

on the plausible assumption that only a very "generalized" transformation could yield answer 1 or answer 5. It sometimes happens that all the nodes are rejected by this heuristic. Then ANALOGY starts over again with a different approach.

Discussion of Node Selection. Node selection as applied to particular nodes without comparison with other nodes is somewhat more straightforward then method selection. Method selection depends on features and node selection does also, but in the case of nodes the ordering or elimination which results applies directly to the node itself, not to an array of methods. The heuristics which have been mentioned for node selection appear to be mainly kinds of (1) difficulty estimate or (2) utility estimate. I tentatively divide them as follows. Ordering of goals by "depth" in SAINT, rejecting harder subnodes in GPS, trying simplest nodes first in LT, are kinds of difficulty estimates. On the other hand, trying closest-to-premises nodes first in the Geometry Machine, terminating search at "dead" positions, rejecting nodes with too many branches, and rejecting figures with an incorrect number of parts in ANALOGY, are kinds of utility estimates.

VIII
The Use of Models

At the end of Section V two types of models, semantic models and analogous (or simplified) models, were mentioned. The fact that the Geometry Machine would not have been able to prove even the simplest theorems without the help of the diagram reminds us of the power of the first type. The value of the second type may be seen by considering the effect of dividing a problem into smaller subproblems. Since the amount of exploration required to solve a problem generally increases exponentially with the length of the solution, replacement of a single large problem by several smaller problems, the length of whose solutions adds up to the same length as that of the original problem, may dramatically reduce the problem difficulty. To quote again from Minsky (1961a):

> In a graph with 10 branches descending from each node, a 20-step search might involve 10^{20} trials, which is out of the question, while the insertion of just four lemmas or sequential subgoals might reduce the search to only $5 \cdot 10^4$ trials, which is within reason for machine exploration. Thus it will be worth a relatively enormous effort to find such "islands" in the solution of complex problems. Note that even if one encountered, say, 10^6 failures of such procedures before success, one would still have gained a factor of perhaps 10^{10} in overall trial reduction. Thus practically any ability at all to "plan," or "analyze," a problem will be profitable, if the problem is difficult.

A Simplified Model in GPS. An implementation of a

simplified model as the source of a plan is found in GPS (Newell *et al*, 1959). The plan is generated by omitting details of the original problem and by solving this simplified problem by simplified methods. The simplification procedure used here ignores (1) differences among connectives and (2) the order of the symbols. The problem, "From (r ⟶ not-p) and (not-r ⟶ q) deduce not-(not-q and p)," becomes "From (pr) (qr) deduce (pq)." The simplified methods solve the simplified problem quite easily, in fact, several solutions are generated. The simplified problems are discarded and only *the sequence of methods* from the simplified solutions are kept as candidate "plans" for the solution of the original problem.

In attempting to apply the plans GPS finds that some of them will not work. One plan which does work, however, is the sequence of methods

1. (A and B) implies A
2. (A and B) implies B
3. (A implies B) and (B implies C) ⟶ (A implies C)

Applying Method 1 and Method 2 GPS obtains "r ⟶ not-p" and "not-r ⟶ q." When GPS tries to apply Method 3 it finds that the second of the two expressions is of the wrong form. Undismayed, GPS sets up a new goal of putting the two expressions in the form required by Method 3. It computes a difference, selects a new method, and successfully produces "not-q ⟶ r." At this point GPS goes off on an (unsuccessful) tangent trying to deduce "not-(not-q and p)" from "not-q ⟶ r."

But finally it applies Method 3 to "not-q ⟶ r" and "r ⟶ not-p" obtaining "not-q ⟶ not-p." The desired answer "not-(not-q and p)" is easily obtained.

An interesting characteristic of the "plans" in this example is that they are extremely non-specific. What they require is only that the form of the inputs for each specified transformation be correct, so that the transformation can be applied. Another point of interest is that a plan may turn out to be illusory -- it may be impossible to carry out. If we generate a large number of plans, must we try out each one totest its validity? And what if using the wrong plan, even once, will be very costly? Clearly there must be some way to test plans without having to try them out. "Plans" on one level may "nodes" to a problem-solver operating on a higher level.

Approaches different from that of GPS may be found in Tonge (1961a), who reduces assembly line balancing problems bit by bit by successive groupings, and in Karg and Thompson (1964), whose program for solving travelling salesman problems first produces a number of sub-optimal solutions and then, having identified easy subdivisions and hard subdivisions by means of the previous solutions, sets to work on the hard ones by themselves.

It may be observed that many of the aids given to heuristic problem-solvers embody models or parts of models used by humans. The Geometry Machine accepts the diagram

without having any part in changing it or improving it. In Samuel's checker-player the terms (features) by which it evaluates positions are given to it. It is true that the checker-player adjusts the weights of the terms itself, but the terms themselves are the product of human models and human ideas of what checkers is about. This is perhaps seen even more clearly in the chess-playing programs where goals such as Center Control are what humans think is important while playing the game.

In Section IV three potential functions of pattern recognition were suggested; briefly, (1) discovering new features; (2) discovering patterns in problem search procedures; and (3) discovering patterns which turn out to be useful models or representations. Unfortunately, as remarked in Section V, no heuristic problem solver to date has made use of pattern recognition techniques to construct new models. It may be that pattern recognition is no more than learning on several levels simultaneously (compare Selfridge and Neisser, 1960). In any case the ability to make effective use of pattern recognition, particularly for the creation of new models, would widen the horizons of problem-solving machines tremendously.

Summary

I Heuristic versus Algorithmic Programs. Heuristic and algorithmic programs are working towards a common end, namely, the development of better problem-solving machines. The two approaches may be regarded as complementary. On the one hand, knowledge of heuristics derived from heuristic programs can provide algorithmic theorem-provers with better ways of eliminating the irrelevant from their proofs. On the other hand, the experience with algorithmic programs can result in simpler reformulations of problems which can also be used by heuristic programs.

II Definitions of "Heuristic." Heuristics seek to use partial or uncertain information during problem-solving. A heuristic is "a method of selection," or more explicitly, "a method which helps in discovering a problem's solution by making a plausible but fallible selection of those alternatives which seem most promising." Heuristic selection is a kind of guessing. There are methods other than heuristic methods which may help in discovering a problem's solution; therefore we insist on an element of guessing or fallibility as part of our definition.

III An Example of a Heuristic Program. The major heuristic in the Geometry Machine of Gelernter is a (simulated) diagram, which gives exactly the same information to

the Geometry Machine as a pictorial diagram would give to a human. The diagram is an interpretation of plane geometry, hence a semantic model of the problem situation. The highly branched problem-solving tree is sharply reduced by elimination of all branches not verified in the diagram. Without the use of the diagram as a heuristic the Geometry Machine would be unable to prove even the simplest theorems in geometry. The Geometry Machine also uses several minor heuristics. It illustrates several basic characteristics of heuristic programs, including: (1) division of problems into subproblems; (2) use of heuristics; (3) recursiveness; and (4) fallibility.

IV Comments on Induction, Learning, and Pattern Recognition. In general, heuristics have an inductive character because they are based on the assumption that certain alternatives may be selected as promising, and others rejected as unpromising, by generalizing on their similarity to other alternatives, in the past, which were promising or unpromising. Generalizing of this kind is also involved in learning and pattern recognition. Three uses of pattern recognition which would be of great potential value in problem-solving machines are: (1) originating new, useful features for characterizing problems; (2) discovering patterns in the proof-search procedure and using them to guide future search; and (3) discovering patterns which turn out to be good models or rperesentations.

V Tree Representations and Levels of Selectiveness. Almost any well-defined problem can be converted into a problem of finding a sequence of transformations leading from a given situation to a desired situation. A convenient representation of the possible sequences is the problem-solving tree (Figure 2, page 21). In checkers, for example, the original node represents the standard beginning checker position. Each node which can be reached in one step from the original node represents a position which can be reached in one legal move from the original position. Nodes which can be reached in two steps represent positions which can be reached in two legal moves, and so on. Legal moves or transformations are called <u>methods</u> and positions or situations which are obtained by application of the methods are called <u>nodes</u>. At any given time the growth reached by the problem-solving tree is a complete representation of the results reached so far in the problem-solving process. Because of the immense size of such trees for interesting problems, trying to find a solution by exhaustive exploration is out of the question. Some <u>selectiveness</u> must be exercised in order to explore only those parts of the tree which appear promising. Three levels of selectiveness, from most specific to most general, may be distinguished.

(1) Selection of which methods to apply to a node already chosen. This may require learning, since the machine may choose methods to apply on the basis of its experience with

previous nodes which were similar. (2) Selection of which nodes to work on out of all the nodes generated so far. The reason that selection of nodes raises serious difficulties is that reviewing the entire problem-solving tree is required in order to make the selection. Since the number of new nodes grows very rapidly (unlike the number of methods), reviewing the tree may quickly become a lengthy and impractical job. An additional complication in complex problems is that the solution of an important subproblem produces a partially modified state of affairs requiring a whole new evaluation. (3) Construction of plans or models of the problem situation. Models permit heuristic selection on an even higher level. Two kinds of model are the _semantic_ model (such as the diagram in the Geometry Machine) and the _analogous_ (or simplified) model. For a given problem, a second easier problem of "essentially the same character" may serve as a simplified model of the original problem, and the solution of the simpler problem can be made to serve as a guide to the solution of the harder one.

VI Heuristics for Method Selection. Several examples of method selection may be given. (1) The LT program of Newell _et al_ uses a _similarity test_ as a heuristic to select which methods (axioms or theorems) have the greatest chance of being applicable. The acceptance or rejection of candidate methods to be applied to a node is based only on that one node. (2) The GPS program, which grew out of LT, selects

methods by computing a difference between two nodes: the present node and the desired node. (If there is no difference, the probelm is solved.) Certain methods are appropriate for reducing certain differences, and it is a logical development for GPS to learn for itself which methods are appropriate. (3) Out of the thirty or so moves possible in typical chess positions, good human chess players consider only a very few. A similar effect has been achieved in chess-playing programs by using "plausible move generators" which propose for consideration of the main program only those moves which promote some particular goal. Elimination-by-not-being-proposed is a heuristic principle used in various heuristic programs. In each of the cases discussed, the selection depends on the features of the node or of several nodes. As an interesting addition to the discussion of method selection, the geometric-analogy machine of Evans shows how new methods may be obtained by "generalizing" on old methods.

VII Heuristics for Node Selection. Several examples of node selection may be given. (1) Samuel's checker-player uses Turing's notion of a "dead" position to help determine which nodes will be explored. A "dead" position is one in which the game is relatively stable. Positions (nodes) which are not "dead" require further exploration. (2) GPS requires that the difference between a newly produced subnode and the desired node be less than the difference between the parent node of the subnode and the desired node. If the

difference is larger, then GPS rejects the subnode. The rationale is that if a subnode is more difficult than its parent node it will not be of much help in problem-solving. (3) The MATER program of Baylor and Simon for finding mating combinations in chess rejects positions (nodes) which increase the opponent's mobility. (4) The geometric-analogy machine of Evans rejects figures which on preliminary inspection reveal a disparity of number of parts added, removed, and matched. Heuristics for node selection appear to be mainly kinds of (1) difficulty estimate or (2) utility estimate. Like method selection, node selection depends on finding good features.

VIII The Use of Models. The Geometry Machine gave an example of a <u>semantic</u> model. The use of a simplified <u>analogous</u> model is illustrated through the solution of a problem in logic by GPS. Working on a simplified version of the problem may produce a number of <u>plans</u> for the solution of the harder problem. Some of the plans may prove illusory but one of them may work -- with great savings in time and effort as a result. If a large number of slightly different plans are generated, a heuristic problem-solver working on an even higher level may treat plans as nodes, and heuristic selection may be needed to choose among different plans. Two additional approaches to planning are the assembly line balancing program of Tonge and the program for solving traveling salesman problems of Karg and Thompson. One conspicuous characteristic of models which actually prove

<u>useful</u> in problem-solving programs is that their essential ingredients were furnished by the human programmers rather than by the machine. It is suggested that greater ability in pattern recognition would greatly increase the power of problem-solving machines, especially with respect to creating their own models.

IX
Conclusion

We can confidently expect that as long as we seek to solve complex problems by machine that there will be continued interest in and experimentation with heuristics. Heuristics lie at the heart of some of the problems which face us on the road to building more powerful problem-solvers. Two of the problem areas in which improvements are much needed are:

1. Implementation of pattern recognition techniques in problem-solvers

2. Better program organization eliminating the many rigidities of present programs

Heuristics are intimately connected with both of these. In regard to problem area 1, pattern recognition and learning are intimately linked with heuristics, as argued in Section IV. A selection made partly on the basis of pattern recognition techniques will usually be a fallible, plausible, heuristic selection. Moreover the origination of heuristics depends on learning and pattern recognition. In regard to problem area 2, an increase in program flexibility will generally allow more choices and more opportunity for heuristic selection. Some of the problems of heuristic selection which arise as a result of allowing the problem-solver greater freedom are suggested in Section V, under "Node Selection". A general conclusion is that heuristics

are likely to be used more rather than less as our problem-solvers become more sophisticated.

No one has yet developed a theory of heuristic. (At least, I have looked hard and have not found any.) Perhaps this is only natural in view of its close connection with so many difficult notions, such as "induction," "plausibility," and "pattern recognition." These ideas are particularly resistant to formalization, and we can expect "heuristic" to be resistant also. Nevertheless in the course of this paper I tried to attack the notion from various angles, seeking to come to grips with it. Although I cannot claim to have been completely successful in this effort, my hope is that these pages give a clearer idea of what "heuristic" means in the context of problem-solving.

BIBLIOGRAPHY

Amarel, S., 1962. "On the Automatic Formation of a Computer Program which Represents a Theory," Self-Organizing Systems, Yovits, M.C., Jacobi, G., and Goldstein, G. (eds.), Spartan Books, Washington, D. C., 106-175.

Baylor, G. W., and Simon, H. A., 1966. "A Chess Matings Combinations Program," Proceedings of the Spring Joint Computer Conference, 28, 431-447.

Berlekamp, E., 1963. "Program for Double Dummy Bridge Problems," J. ACM, 10, 357-364.

Bernstein, A., et al., 1958. "A Chess-Playing Program for the IBM 704," Proceedings of the 1958 Western Joint Computer Conference, 157-159.

Cooper, D. C., 1966. "Theorem-Proving in Computers," Advances in Programming and Non-Numerical Computation Fox, L. (ed.), Pergammon Press, New York, 155-182.

Davis, M., 1963. "Eliminating the Irrelevant from Mechanical Proofs", Proc. Symposia in Applied Mathematics, Vol, XV, American Mathematical Society.

Ernst, G. W., and Newell, A., 1966. Generality and GPS, Doctoral Dissertation, Electrical Engineering Department, Carnegie Institute of Technology.

Evans, T. G., 1964. "A Heuristic Program to Solve Geometry-Analogy Problems," Proceedings of the Spring Joint Computer Conference, 25, 327-338.

Feigenbaum, E., and Feldman, J. (eds.), 1963. Computers and Thought, McGraw-Hill, New York.

Gelernter, H., and Rochester, N., 1958. "Intelligent Behavior in Problem-Solving Machines," IBM Journal of Research and Development, 2, No. 4, 336-345.

Gelernter, H., 1959. "Realization of a Geometry Theorem-Proving Machine," Proceedings of an International Conference on Information Processing, Paris, UNESCO House, 273-282. Reprinted in Feigenbaum and Feldman, 1963.

Gelernter, H., Hansen, J.R., Loveland, D. W., 1960. "Empirical Explorations of the Geometry-Theorem Proving Machine," Proceedings of the Western Joint Computer Conference, 143-147. Reprinted in Feigenbaum and Feldman, 1963.

Gilmore, P. C., 1960. "A Proof Method for Quantification Theory," IBM Journal of Research and Development, 28-35.

Karg, R. L., and Thompson, G. L., 1964. "A Heuristic Approach to Solving Traveling Salesman Problems," Management Science, 10, 225-248.

Kuehn, A. A. and Hamburger, M. J., 1963. "A Heuristic Program for Locating Warehouses," Management Science, 10, 643-666.

Marill, T., 1963. "A Note on Pattern Recognition Techniques and Game-Playing Programs," Information and Control, 6, 213-217.

Mesarovic, M. D., 1965. "Toward a Formal Theory of Problem-Solving," Computer Augmentation of Human Reasoning, Sass, M. A., and Wilkinson, W. D. (eds.), Spartan Books, Washington, D. C., 37-64.

Minsky, M., 1956. "Heuristic Aspects of the Artificial Intelligence Problem," Lincoln Laboratory, MIT, Lexington, Mass., Group Report, 34-55, ASTIA Document No. 236885.

Minsky, M., 1961a. "Steps Toward Artificial Intelligence," Proceedings of the IRE, 8-30. Reprinted in Feigenbaum and Feldman, 1963.

Minsky, M., 1961b. "Descriptive Languages and Probelm-Solving," Proceedings of the Western Joint Computer Conference, 215-218.

Minsky, M., 1966. "Artificial Intelligence," The Scientific American, 215, 247-260.

Newell, A., and Simon, H. A., 1956. "The Logic Theory Machine," IRE Transactions on Information Theory, Vol. IT-2, 61-79.

Newell, A., Shaw, J. C., and Simon, H. A., 1957. "Empirical Explorations with the Logic Theory Machine," Proceedings of the Western Joint Computer Conference, 218-239. Reprinted in Feigenbaum and Feldman, 1963.

Newell, A., Shaw, J. C., and Simon, H. A., 1958a. "Chess Playing Programs and the Problem of Complexity," IBM Journal of Research and Development, 320-335. Reprinted in Feigenbaum and Feldman, 1963.

Newell, A., Shaw, J. C., and Simon, H. A., 1958b. "Elements of a Theory of Human Problem-Solving," Psychological Review, 65, 151-166.

Newell, A., Shaw, J. C., and Simon, H. A., 1959. "Report on a General Problem-Solving Program for a Computer," Proceedings of an International Conference on Information Processing, 256-264, UNESCO, Paris, Reprinted in Computers and Automation, July, 1959.

Newell, A., Shaw, J. C., and Simon, H. A., 1960. "A Variety of Intelligent Learning in a General Problem-Solver," Self-Organizing Systems, Yovits, M. C., and Cameron S. (eds.), Pergammon Press, New York, 153-189.

Newell, A., 1962a. "Some Problems of Basic Organization in Problem-Solving Programs," Self-Organizing Systems, Yovits, M. C., Jacobi, G. T., and Goldstein, G. D. (eds.), Spartan Books, Washington, D. C., 393-425.

Newell, A., 1962b. "Learning, Generality and Problem-Solving," Proceedings of IFIP Congress 62, North Holland Publishing, Amsterdam, Holland, 407-412.

Newell, A., 1965a. "Limitations of the Current Stock of Ideas about Problem Solving," Electronic Information Handling, Kent, A. and Taulbee, U. (eds.), Spartan Books, Washington, D. C., 195-208.

Newell, A., and Ernst, G., 1965b. "The Search for Generality," Proceedings of IFIP Congress 65, Kalenich, W. A. (ed.), Spartan Books, Washington, D. C., 17-24.

Polya, G., 1954. Mathematics and Plausible Reasoning (2 vols.), Princeton, N. J., Princeton University Press.

Robinson, J. A., 1965. "A Machine-Oriented Logic Based on the Resolution Principle," J. ACM, 12, 23-41.

Samuel, A. L., 1959. "Some Studies in Machine Learning Using the Game of Checkers," IBM Journal of Research and Development, 3, 211-229. Reprinted in Feigenbaum and Feldman, 1963.

Selfridge, O., and Neisser, U., 1960. "Pattern Recognition by Machine," The Scientific American, 203, 60-68.

Simon, H. A., and Newell, A., 1958a. "Heuristic problem-solving," Operations Research, 6, 1-10.

Simon, H. A., Newell, A., and Shaw, J., 1958b. "The Processes of Creative Thinking," Rand Corporation Paper P-1320. Reprinted in Contemporary Approaches to Creative Thinking, Gruber, H. E., Terell, G., and Wertheimer, M. (eds.), Atherton Press, New York, 63-119, 1962.

Simon, H. A., and Simon, P., 1962. "Trial and Error Search in Solving Difficult Problems: Evidence from the Game of Chess," Behavioral Science, 7, 425-429.

Simon, H. A., 1963. "Experiments with a Heuristic Compiler," J. ACM, 10, 493-506.

Slagle, J., 1961a. A Computer Program for Solving Problems in Freshman Calculus, Doctoral Dissertation, Massachusetts Institute of Technology.

Slagle, J., 1961b. "A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus," Computers and Thought, Feigenbaum and Feldman (eds.), 1963.

Stefferud, E., 1963. "The Logic Theory Machine: a Model Heuristic Program", RAND Corporation, Santa Monica, California, Rm-3731-CC, 187 pages.

Tonge, F. M., 1960. "Summary of a Heuristic Line Balancing Procedure" Management Science, 7, 21-42. Reprinted in Feigenbaum and Feldman, 1963.

Tonge, F. M., 1961. A Heuristic Program for Assembly Line Balancing, Prentice Hall, Englewood Cliffs.

Tonge, F. M., 1963. "Balancing Assembly Lines Using GPS" Symposium on Simulation Models: Methodology and Applications to the Behavioral Sciences, South-Western Publishing Co., Cincinnati, 139-151.

Travis, L., 1964. "Experiments with a Theorem-Utilizing Program," Proceedings of the Spring Joint Computer Conference, 25, 339-358.

Uhr, L., and Vossler, C., 1951. "A Pattern-Recognition Program that Generates, Evaluates, and Adjusts Its Own Operators," Proceedings of the Western Joint Computer Conference, 19, 555-570. Reprinted in Feigenbaum and Feldman, 1963.

Wang, H., 1960a. "Toward Mechanical Mathematics," IBM Journal of Research and Development, 4, 2-22.

Wang, H., 1960b. "Proving Theorems by Pattern Recognition I," Communications of the ACM, 3.

Wang, H., 1961. "Proving Theorems by Pattern Recognition II," Bell System Technical Journal, 40.

Wos, L., Carson, D., and Robinson, G., 1964. "The Unit Preference Strategy in Theorem Proving," Proceedings of the Fall Joint Computer Conference, 26, 615-621.

VITA

Darien Adams Gardner, son of Fentress and Emily Elizabeth Gardner, was born in Lake Wales, Florida, on April 12, 1942. He was graduated from Princeton High School in Princeton, New Jersey in June 1960 and received a Bachelor of Arts Degree from Haverford College in June 1965. After graduation he spent one year in India on a Fulbright Fellowship.