July 4, 1956

A FRAMEWORK FOR AN ARTIFICIAL INTELLIGENCE

By Marvin Minsky

The following is a general description of certain components of a machine which is expected to work at the solutions of problems in a reasonably intelligent manner. Its construction is not imminent — the system described here is intended only to provide an heuristic framework in which certain general procedures can be examined in some detail. The machine is described in terms of a diagram with several "blocks." None of these blocks are described in anything approaching complete detail; on the contrary, each of them raises its own set of theoretical problems. The value of this approach, I feel, is that such a framework makes it possible to work in the direction of relatively hard solutions to specific problems. The danger is that of being trapped into making unnecessarily elementalistic distinctions in an effort to maintain the boundaries of the boxes. But this may be a smaller risk than that involved in evading separation; then one is liable to find that one has n boxes, n-1 of which contain a few relays or valves, and the last behaves intelligently! I claim, at the least, that no one of my boxes will behave intelligently.

The idea of constructing the framework, and many details concerning the "Characterizer," the "Method box," and the "Clean-up box," originated or were agreed upon in discussions between the author and John McCarthy. Some details of the "evaluator" originated in discussion with Ray Solomonoff. Many ideas came out in meetings of the Artificial Intelligence Group as a body.

For concreteness, I will often talk as though we had a particular interest in building a machine to find proofs for theorems in mathematical logic. Actually, the machine is intended to operate over much broader classes of problems, including the making of empirical perceptions and judgments. The use of the logic-proving example is dictated not only by the convenience of the clarity of the problems involved, but by the

inspiring progress of Newell and Simon, and of More, in their independent designs for machines to solve problems in this category. The members of the Artificial Intelligence Group have been particularly interested in these developments, which have had considerable effect on the direction of our thinking. With a little effort these models can be made to serve as examples of some of the parts of the present model.

Before outlining the machine, it will be helpful if the sympathy of the reader can be enlisted in regard to certain lacunae in the presentation. In fact, without it, the text will be incomprehensible. There are a number of terminological ambiguities, and there is one major one. At each step in the course of solving a problem, one can look back (if one has a memory) at a sort of exploration process in which at various times in the past one of a number of alternative actions was taken. The totality of such branching decisions I call the "exploration graph." It is assumed that a description of this structure is stored somewhere in the machine. Attached to the "vertices" (branch points) and "links" of this graph are certain data, such as the length of time that has been spent on each link. Also, with each vertex of the graph must be stored a description of what that point means, i.e., what problem would face you if you were at that point. Without such data the graph would be meaningless. Each link descending from a vertex represents a proposed step that has been considered as a possible next step to be taken in solving the problem represented by the vertex above. I have called the vertices of the exploration graph "situations," or occasionally, "subgoals." The art of problem-solving could be regarded as the art of efficiently manipulating exploration graphs, although I will not maintain that this would be an illuminating viewpoint.

Now the difficulty is that there are liable to be several different kinds of vertices on an exploration graph, and I have taken no trouble to distinguish them in the exposition. To take logic-proving as an example: In one kind of situation our problem may be simply "find a proof for this theorem: ... ."

Assuming we fail to do this our situation may now take the form, "well, then, find another theorem which might be helpful in proving the one you just failed to prove directly." Or, "find a theorem in your memory that is like (in a well-defined sense) the one you are trying to prove, and see how you proved that theorem." In general, I have tacitly assumed that the typical situation will be like the first above. The general "situation", as I intend ultimately to use the term, will be one or more expressions in a language, plus a well-defined problem concerning them.

Again, I will use the term "Method" as though it meant "particular method for proving a theorem." But there are several kinds of methods, and again for heuristic reasons I do not distinguish them explicitly. Each kind of situation calls for a different kind of method. There is a kind of Method used in trying to find a proof for a theorem, a different kind for trying to find a suitable subgoal. It is the application of a Method to a situation that produces a new situation, and thus that the exploration graph can grow. But it does not require application of a "Method" to enable the process to "move around" on the existing part of the graph; this process will be controlled by the device V to be described below.

Another terminological point is that I do distinguish between two kinds of "links," namely "established" links and "unestablished" links. This distinction is carried upstairs from the logic-machine example, and is intended to reflect the idea of "subgoal" as used by Newell and Simon. The idea is that when one has a large gap in an argument, it is a good thing to establish an island that will, one hopes, leave two smaller gaps. Thus one creates more gaps than there were before. On the other hand, one also can "fill" a gap by "establishing" a chain of reasoning that crosses it. Thus some Methods create "unestablished" links, some create "established" links, and some serve to "establish" links that were not established before. And some Methods insert a new vertex into an unestablished link producing two of the same where there was one before. I have carried this

notion over to the general plan because I believe that one uses something like this notion of "provisional" vs. "established" in general problem-solving; even when there is no such decisive tool as logical validation, one does assign degrees of conviction to the various ways one has of justifying arguments, inductions, and judgments. In logic there are only two degrees, and we should perhaps leave room in our graphs for a more delicate measure of validity of a transformation between situations.

. . . . .

We will start by outlining the behavior of the basic units.

. . . . .

PROBLEMS. First, there is a set of problems to be solved; these are not part of the machine and are not available to it in advance of the order in which they are presented to it, say, by some other machine or the environment. One may think of a "problem" as a series of symbols fed into the input of the machine; this series, or "string," is to be thought of as a sentence or question in some known language. The problem, once in the machine, is first carried to a relatively simple unit "LC."

LC. THE LOGICAL AND SYNTACTIC CLEAN-UP UNIT.

This unit has a fixed program which takes the submitted string and subjects it to a series of operations which bring the given string, so far as possible, into one of a number of standard forms. Failing this, it attempts to do the same for parts of the string. Each of the transformations are to be in the direction of simplicity, and none are to change the meaning in any way. The idea is to remove detectable redundancies, introduce standard abbreviations wherever possible, and in general reduce the load of mechanical transformations that might otherwise burden the more sophisticated parts of the machine. An example for logic would be the routine replacement of "$\sim\sim x$" by "x" (always provided there is no immediate reason not to do this). McCarthy has pointed out that such a device will raise

the overall efficiency of the machine appreciably if it is applied not just to the incoming problem, but routinely to expressions occurring throughout the overall search process.

CH. THE "CHARACTERIZATION" COMPUTER

CH is one of the basically important units of the machine. It is here that the processes of "abstraction" can be most clearly localized. At least, it is here that a basic type of abstraction is formed. Briefly, CH divides its inputs into certain categories, and associates with each input the name of its category. This name is called its "characterization."

The discussion deserves some motivation here. Suppose we have a machine which has a history of problem-solving. At the risk of being elementalistic, assume we have examined its past behavior and agree that we can discern that it has a certain finite set $M_1, \ldots, M_n$ of procedures which are the "Methods" it uses to solve problems. Now we are interested only in machines that make effective use of their past experience. We will certainly be interested in, among others, these aspects of the machine's behavior:

i) How does the machine decide which methods to use on a given problem?

ii) In particular, if it used prior experience on new problems it must use some measure of similarity of the new problem with old ones, and presumably first try methods that have proved to be successful on similar old problems.

iii) How does it find new methods, or make new ones from combinations of old ones?

Aspect ii) would seem to take priority and in fact CH is the device which handles, at the immediate level, this "similarity" problem. I will describe CH abstractly: one could discuss specific schemes for CH at great length (and I intend to elsewhere soon), but a brief outline will be sufficient here.

CH is simply a computer which computes, for each input S, the values of a finite set $C_1(S), \ldots, C_n(S)$ of functions.

Each function C_i is called a "character" of CH; each C_i can have only a finite set of distinct integral values. Thus each C_i divides all possible inputs into a finite number of sets: the value of C_i tells us which set the input belongs to. The sets defined by the different C_i's may or may not have anything to do with each other. The heuristic is that the value of each C_i tells us something about the input; it is hoped that what it tells us will be useful in solving the problem. (In particular, it would be nice if we obtained information that could be interpreted to help us decide what should be our next step! This interpretation is the job of the remainder of the machine.) We will define the "characteristic"* of an expression S (an input to CH), as the ordered string of integers $C_1(S), \ldots, C_n(S)$, and denote the "characteristic of S" by C(S). The characteristic of an expression itself defines a set of inputs, namely, the intersection of the sets defined by the individual characters. It is hoped that the characters are such that the characteristic tells us a good deal about the expression. But it is also hoped that C(S) will spare us a lot of unnecessary details about S and give us mostly information that will be useful in solving the problem!

Although metaphysicians may desire something more subtle, we will tend to associate the values of the characters with the sets defined by the inverse functions, and regard them as "abstractions." The sets may be regarded as equivalence classes with respect to the "property" defined by the characters, or you may choose another formulation. If the "property" so defined has no intuitive meaning, or other practical significance, then some might refuse to admit it to the category of "abstraction;" I would simply prefer to regard it as a "poor," "useless," "unnatural," etc., abstraction.

In any case, it will be seen that our machine is so programmed that, to the extent that the characters of two expressions agree (and the way in which this agreement is measured involves a learning process which is servoed by success), the machine will try to treat them in the same manner, that is, with

* I have not been consistent below in distinguishing "character" and "characteristic."

the same Methods. When the results of such treatment yield expressions which are sufficiently different to reflect a character change, then the methods will also diverge. The agreement of characters will provide the measure of similarity by which the results of prior experience are utilized. Obviously the "quality" of the characters will affect the ability of the machine to use its experience, and if we can't think of some real good a priori characters for our machines, we will have to install a character generator and success servo to obtain some for us. We will return to this later. A few remarks about characters are in order.

1. The efficiency aspect. Since the character of an expression is supposed to give us information that will be useful in solving the problem, and to suppress information irrelevant to that goal, a good characterization process should yield expressions smaller than those fed into the CH. Again, if the characters are good, then it will be useful to store results like "Method M_i was very effective in solving a problem of type (character) C." Now in fact, I would expect good characters to achieve a very large compression in space, perhaps by several orders of magnitude.

2. Relevancy. The character is supposed to divide incoming problems into classes to make it easy to decide how to approach a solution. In a logic machine which is supposed to establish deductive chains, the characters would be most effective if they divided problems into classes with relevance to their deducibility. They must also match the methods available to the machine. Note that "interesting" general deducibility results might be quite useless unless they were properly related to the other resources of the machine. We will discuss below this relation between the characters and the Methods. For example, the numerical characters used by Newell and Simon in their "matching" and memory search routines apparently divide expressions into moderately useful classes. When the relatively trivial substitution tech-

niques (which are selected by the simple formal numerical characters) fail, the machine then proceeds to use methods in which the "characters" are things like "the numerical description of the antecedent of the main connective." If one will accept the interpretation that the machine uses these objects as "characters" in my sense, this provides an example of characters which relate to the deducibility properties of the sentence. (That is, a character which describes a property of the "antecedent" of a logical expression is clearly a candidate for having relevance to deducibility.)

3. Types of categories. (Both McCarthy and Solomonoff have something to say on this subject, and I have benefited from discussion with them.) Provisionally, we can describe a few kinds of characters, in terms of the properties or categories they define. I confine the discussion to expressions in a language; I hope to add a discussion concerning categories of sense data and evidence later.

i) Grammatical categories: Purely formal properties of expressions. In particular one might use the new analytic structure described by N. Chomsky, and characterize an expression in terms of the "primitives" and "transformations" that generate a sentence in the given language.

ii) Syntactic categories. Classifications of the logical structure of the expressions.

iii) Semantic categories. For example, you can decide whether the question relates to mathematics or chemistry (which may require different methods) by classifying the nouns into semantic categories.

A few special types which may be of interest:

iv) Mnemonic categories. E.g., is there a theorem in the memory which resembles the given one in such and such a respect?

v) A character-generating character which might be useful:

"Compare the given expression S in some (specified) way with that theorem T in the memory which has up to now been used most effectively. (Assume such a record is kept.) Let the new character C(S) have value 1 if they are similar, as specified, 0 otherwise."
Characters generated in such a way would seem to have a reasonably good a priori chance of being useful.

vi) Special deducibility category.

"Can the expression S be deduced from a particular given expression T in less than (given) n time units?"
(I doubt if this particular type would be very powerful, but if a machine were set to work on very difficult problems in a limited domain, highly special categories might be natural and appropriate: A trained specialist may need very specialized tools.)

vii) "Second characterization."

There is some danger that, since we want the characters to be small in size, we risk loss of important information. That is, it may be the case that two expressions S and S^* differ in some way important to us, yet $C(S) = C(S^*)$. However, this will not result in the machine blindly treating both in the same manner. (I must anticipate the next section here.) It is true that the machine will perform the very next step identically for both. But if the difference between the expressions is really important, the difference in machine behavior should not be long in appearing. Suppose, in fact, that the next move of the machine is to apply some method M_j to transform the situation. While M_j is determined by $C(S) = C(S^*)$, this does not mean that the results $M_j(S)$ and $M_j(S^*)$ will be equal. This suggests a wholesale way of producing new characters with an excellent chance of yielding new information relevant to the rest of the machine processes. Select any transformation that the machine may be able to use, say M_j. Then for any character C_o, we obtain a new character $C_o^{M_j} = C_o(M_j(S))$. [This information can also be derived by extending the Method search routine.]

MM. THE METHOD-MEMORY UNIT

At each stage in the problem-solving process we have a "situation." In the theorem-proving process a situation could be described roughly by stating

1. A list of expressions.
2. A lattice or graph showing the relation of the expressions in the situation, i.e., the theorem to be proved, the relation of the various subgoals and hypotheses, and the state of the proofs of each.
3. An indication of which point of the graph is currently under attack.

In any situation we want to compute what should be the next step to try. This computation will be made by the "situation evaluator V" which will be described shortly. Suppose that it has been decided that the most valuable next move would be to try to prove some particular theorem T. Then T is fed into the characterizer CH and the resulting output of CH, namely C(T), is fed into the Method-Memory Box MM.

The machine is provided at the start with a finite collection of "Methods" $M_1, \ldots, M_n$, each of which may be a large subroutine which, when presented with a situation, yields a new expression or set of expressions. We will assume for the moment that this set is fixed: later the problem of improving the Method collection may be considered. The Methods are the set of devices with which the machine can convert one situation into another. Now the method memory box MM is designed to answer the following kind of question: "Given an expression of character C(E), what is the best method to apply first in finding a proof of that expression?"

Thus the MM unit, when given a string (a "characterization") from CH, yields as output a series of indices representing methods to be applied, in order of preference. This recommended order is the result of a basic learning process. For each character, we store in MM a record of the past effectiveness of each of the methods M_i in proving theorems of that "character." We

will return to the question of efficient ways in which the machine uses its experience to get this data.

But, even before the method box can be consulted, we have to select an immediate problem or subgoal. Now at any moment the system is considered to be located at some point of what we call the "exploration graph." Descending from this point are a number of "links" and we want to select one of these. (Each link may connect with, e.g., a sub-theorem whose proof might be useful.) We want a selection process which will evaluate these alternatives, and pick the most favorable. This selection process must also be equipped with a "favorability threshold" which will direct it to move up higher on the graph in the case that none of the immediate alternatives are sufficiently attractive.

This selection is done by a device called the "situation evaluator," V. The judgments of V may be relatively crude for two reasons. First, as in machines to play combinatorial games, the situation value functional is magnified in virtue according to the depth of the alternate strategy graphs to which it is applied. Second, as will be seen, the functional we have in mind will be heavily weighted by the amount of time that is spent on each branch of the exploration graph: thus, if a judgment is poor, some time will be wasted, but before long V will change its rating and move to another branch. The use of TIME as part of the rating of the merit of various procedures will play a large part in the operation of V as well as in the learning behavior of the method box MM. It may be that use of "time" in such a way is an important aspect of "intelligence." I will write up my idea of how V ought to operate in more detail later, and just outline it here.

The exploration graph of the problem shows the relation of the various subgoals that have been considered up to the present. Each segment of the graph has associated with it an entry showing how much time has been spent on trying to "establish" the logical link represented by the segment, and also an entry showing whether the link has in fact been established. For bookkeeping reasons it will also be convenient to associate with each vertex an entry

showing the total time spent to date on the subgraph below that vertex. (If the graph is reentrant, the time has to be adjudicated in some fair manner.) Then, "other things being equal," when there is a choice of which way to go, the system should choose the branch which has received least attention.

Now V is to be equipped with a learning program which will sharpen up these judgments by making a prediction about the difficulty of each of the proposed alternatives! This prediction will take the form of an estimate of how long it will take to establish a given link. V will then add this estimate to the entry showing the time already spent on the link, and use the sum for comparison with those obtained for other alternatives.

The estimate is obtained as follows. Whenever any link is completed (the simplest example is when a statement S_1 is shown to be deducible from S_2), we store in V a record which associates the (pair of) characters of the vertices of the link with the length of time spent on establishing the link. If a link is uncompleted at the end of the problem, a similar entry is stored showing the length of time wasted. Thus V will contain a matrix (just as MM does) where the rows and columns are characters, and each entry carries data showing how long it has taken in the past to establish a link between pairs of statements whose characters index that entry.

Now this matrix will be much smaller than the matrix one would need if one were to try to remember the actual situations involved rather than just their characters. It is quite clear that the information supplied by V will be especially useful if the characterization process CH is realistic, i.e., if the equivalence classes of situations created by the inverse of the CH operation truly reflect an equivalence with respect to the methods used by the machine to solve the problems the machine is to meet. On the other hand, even if this is not the case, it will still be worth while using V's advice (if any), for the learning process of V will serve to discourage the spending of excessive time on situations whose characters are not reasonably decisive (with respect to getting a clear response from M). When problems of

a given character resist efficient solution, we must always try to see whether this is because i) the machine's method resources are not powerful enough for problems of that character or ii) the given character unfortunately groups problems together which individually require different methods, making it impossible for MM to converge on an efficient method choice. In the latter case the machine will have to operate at random, and this can never be very efficient.

There is an administrative problem concerning what proportion of the machine's total effort should be spent on actually trying to fill gaps ("establish links") and what proportion to spend on the surveying and estimating the difficulties of alternatives. By definition, it would be bad to spend too much effort in either process. (The brute force method has the dubious theoretical advantage that it might sometimes eventually solve the problem; the all-survey method wouldn't, although in principle it might indicate the exact steps to be followed in a proposed solution.) If we wish to refine V, here are a few ways in which it can be done. V as described so far contains only a storage matrix with entries derived from past experience, plus the equipment for using this information to make decisions as to how to move in the exploration graph. If there are a large number of possible character values*, then it will often be the

- - - - - -

* Note: When we think of constructing a machine from scratch, or of programming a machine of the type described here, with provisions for improving the set of Characters and/or the set of "Methods," we have to make certain executive decisions. Is it good to have a great many characters, the better with which to distinguish different kinds of problems? Does it always improve matters to add to the supply of methods, if there is a risk that they do not match the characters, and will result in unnecessarily large search times? The range of solvable problems might be slightly increased at great cost in efficiency. (The optimal state would be that in which there are exactly as many character values as methods, with 1-1 pairing of character with optimal method. But I doubt if this is a useful goal to look for in thinking about machines where either CH, MM, or the problem range is subject to change.)

case that there is no entry yet made in the V matrix when it is consulted concerning a new link. Here are some ways in which we might obtain an estimate for the missing entry.

i) The trivial approach is to start the machine out on the link of interest and see how long it takes to close it. Let us suppose, however, that the class of problems on which the machine is working is known to be so difficult that it has been decided to program the machine not to go ahead on any link until an estimate has been provided for the feasibility of closure. This corresponds to the idea of not trying anything until you have some idea, however tenuous, of your chances of success and/or your expected effort. Note that this is not as strong a program restriction as it might at first appear, since, for genuinely routine links, V will be sure to contain an entry, presumably an encouraging one.

ii) Another (non-trivial) way to compute an entry for V is to replace the given problem by a simpler "sample" problem of the same character and see how long it takes to solve it. (As a by-product one could store a record of the methods that prove successful in the sample problem, in the reasonable hope that they will be useful in solving the real problem should V decide to go ahead on the real problem. This information could also be fed to MM as part of the regular learning routine for MM.) Now it seems to me that this is a very important aspect of the way humans work on problems. To see if a certain approach is worth applying to a given (difficult) problem, we often test the approach first on a simpler problem which seems to us to have some of the features of the given problem. Now in many cases the sample problem is chosen not on the basis of its actual usefulness in some step of the real problem, but on what might be called its "apparent similarity." That is, it is selected with regard to a combination of its "characterization" features, and also with regard to some "simplicity" feature. The "sample problem" might in fact be called a "model" problem; I feel that one important inter-

pretation of the term "model" can be described in terms like "character-preserving," always with respect to whatever "characters" (i.e., "abstractions") we have in mind at the moment.

The point is that this substitution of a "model" or "sample" problem for the given one is a step which is essentially different from the process we have been calling "construction of a subgoal." The distinction cannot, I think, be extended very generally, but there is a clear difference for, e.g., the logical-theorem-proving machine. It is simply that a "subgoal," if all "gaps" are filled, will appear as an expression in the final proof. A "model" problem will not (except by lucky accident). What will appear, however, if the modeling process was meritorious, is the sequence of methods.

How can we construct useful "model" problems? It would not surprise me if this process will have to occupy another fundamental black box for our machine, and pose another major problem in artificial intelligence theory. For I visualize an intelligent machine as well equipped with a set of devices representing various models, and that, in the case of humans, these represent a most important part of its endowment. They represent a large part of that which is laboriously pumped into each child by its culture.

We should be cautious in attempting to design a machine all of whose talents are evolved. Just because the human brain does not, evidently, have all the machinery for adult intelligence initially "built in," it need not, therefore, be regarded as having to "evolve" all this equipment. One might present rather an extreme alternative; the brain need only contain a simple "machine-assembling" machine and a good supply of raw materials. The intelligent environment then supplies the program which causes the simple assembling machine to construct a thinking machine of very high complexity. The program, and the coding for the program, is stored in the "culture," particularly in the family and educational aspects, and is stabilized there by evolutionary mechanisms. Now such a view would be too elementalistic, but I think most views go much too far in the other direction. I

suspect that, with a moderate endowment of Methods, Characters, and Models, a machine could attain an astonishingly high intelligence just through the evolution of its storage matrices in V, MM and the model assigner. (To obtain human-like intelligence one would need to add various contingency-learning processes, perhaps both associative and reinforcement in character.) Evolution of, say, the Characters would increase its ability, and the same would hold for Methods and Models. But perhaps for humankind, much of the evolution of these "conceptual and operational vocabularies" is done by the culture as a whole, and the amount of evolution in, say, finding new Methods, is not great for most indivlduals. The argument is much too elementalistic to be worth defending. An equally compelling view is that each man does in fact perform a great deal of evolution over these vocabularies; what is rare is the event in which a man can code his discovery in such a way that he can instruct others to build in their brains the useful device he has found.

ii) continued. I don't want to describe the contents of any particular "model box" because those that I could describe are tied down to very mundane special examples. But I would like to point out that it may be possible to construct a rather abstract, general purpose, model for machines of the sort considered here! My idea is based on a conjecture about the structure of the mapping induced on the characters of CH by the operators of MM. This mapping is, in general, unpleasantly many-many. My conjecture is that, as evolution goes on and (I am assuming that we are searching at least character space and perhaps method space) the machine becomes more and more skillful at, say, proving theorems, the mapping will tend to become many-one, for each method M. Each method M effects (this is all heuristic) a transformation within the set of situations; if the matching of the characters and methods is meritorious, this will induce a reasonably unique map within the set of characters, and this map will be reasonably close to a homomorphism of the map on the situations.

This conjecture is almost true by definition, since, if the effect of the methods cannot be predicted through the character memory of MM, the characters can't be very useful.

Once this good situation is approximated, we automatically come into possession of a model which looks like it would be very powerful. I will call it the character algebra; it consists of a tabulation of the maps induced between the characters by each of the methods. Now if we are presented with a situation S_1 and are asked to derive situation S_2 (e.g., to show that theorem T_2 is deducible from T_1 and the axioms), we simply replace S_1 and S_2 by their characters C_1 and C_2. Then we look in the character algebra for a string of M's which carry us from C_1 to C_2. This search for such a string of M's in the character algebra is a sort of trivial reflection of the search that normally would be carried out in the full exploration graph of the problem, but each step here is just a table look-up. To facilitate application upstairs, it would be useful to store, along with the table for the character algebra, data on the time each transformation step usually takes. Then the search in character algebra can be programmed to find a string of M's satisfying the conditions, while making a reasonable effort to minimize the sum of times along the string. Then going upstairs, the methods of the string are applied successively to S_1. If all goes well, we will finally obtain S_2 or some S_2' with the same character. Then the character algebra will have served the role of a speedy, useful model. If all does not go well, then one must find a subgoal as usual. Having found a subgoal, the character algebra can be tried again. One could hardly expect to solve complex problems by this method in a single application, but there ought to be a level of complexity between steps for which the method would be efficient. Since V is equipped with a crude measure of difficulty across links, it could be used to decide when the character model should be tried.

In conclusion, I ask the reader to temper his annoyance at the imprecision of the exposition and to try and see in it what I think may be an extremely powerful heuristic. For, given a set

of characters and a set of methods (and assuming that the operations are defined on a space which contains the range of the M's) the character algebra is obtained by a simple mechanical routine, and generates a useful "model" of the problem.

iii) Returning to the problem of V obtaining an estimate for the difficulty of a given link, we observe that at any time it ought to be possible to arrange the characters in a rough lattice according to their discriminative power. Then if we temporarily ignore those of relatively delicate discrimination, we obtain a smaller set of the coarser characters. Now we can contract the "difficulty" matrix stored in V by removing the rows and columns of the delicate characters and adding their entries to those of the appropriate coarse categories. There will now be a better chance that the square of the contracted matrix corresponding to the link in question will contain an entry. This will provide a crude estimate.

In any case it might be a good thing to have some ranking of status of the different characters, and to maintain a contracted matrix for the most effective coarse discriminations. If the characters are being rated and selected this will have to be done anyway.

The description of V should include a brief outline of the bookkeeping for moving around the exploration graph. V should be programmed to keep to a minimum at all times the maximum of the sums in each descending chain of the estimated difficulty of all unestablished links of the chain. It should also place some demerit weighting on the total length of each chain, because very long arguments are less preferable than short ones. Each vertex of the graph should have a time register; this register should at all times show the sum of unestablished links from that point to the top. An occasional clean-up routine will be required to keep this up-to-date in the case that higher links get established. I expect that it will turn out that there is a simple recursive procedure by which V can accomplish this in its wanderings with the aid of a very few registers and negligible loss of time.

THE MASTER PROGRAM AND THE LEARNING ROUTINES.

The overall operation of the machine will be under the control of a very simple master program. This program has to recognize when the problem has been solved (we assume that the problems are well defined, or that some human is given the job), or when too much time has been wasted. The latter might as well be set by an arbitrary time limit, or the time limit can be presented as part of the problem.

Now how do we learn from the solution? My inclination is that if the machine fails to solve the problem, it might be a good idea not to try to salvage anything. If this turned out to be too wasteful, one could program a SURVEY UNIT to examine the final exploration graph. (This might turn out to be a basic complicated block, but I tend to doubt that it will have to be very profound.) For the failure case, the SURVEY UNIT finds on which links, and below which vertices, the most time was wasted. Demerits for the case of time wasted on links would be registered in the appropriate entry of the V matrix. Also the MM matrix should receive demerits for time wasted on links. It isn't particularly obvious to me where to place demerits for vertices below which excessive time was wasted. One way to prevent recurrence would be to store the demerit in MM indexed by the situation-method pair which produced the unlucky vertex.

A more powerful device that ought to be installed is the following: Adjoin to the method box a new method M*. Now whenever the system is some vertex, X, and M* is called forth from the method box, the effect of M* is to return the system to wherever it was just prior to X. M* further restricts it so that whatever happens next, the system cannot come right back to X. This is fairly foolproof since the minimax time program will stop any oscillation after a short time. To complete the scheme the survey unit is programmed as follows: Whenever a particular vertex has proven particularly unlucky, score a positive (!) merit rating in MM in the entry indexed by its characteristic and by the special M*. The result of this subroutine will be that the machine will learn not to waste its time on expressions having

that character. This is a valuable kind of learning, for it enables the machine to use characters which can indicate situations in which the machine is unlikely to be successful -- where its Methods are inadequate -- and to avoid, if possible, wasting time in such situations.

In the success case, the SURVEY UNIT again records demerits for time wasted, and records virtue points for successful links, and also for vertices. In the success case there is no analogous need for a special operator; one stores in MM virtue points for the choices which lead to and from each vertex.

The master program will also run periodic clean-up routines over the two learning systems V and MM. This program will handle the method of combining the new data with what has been preserved of the old so as to arrive at revised preference orders for MM and revised "difficulty" estimates for V. Exactly how this is done is a matter for the "learning theorist." The exact form of the functions involved should not be very critical because of their position in the learning servo, and particularly because, if the functionals are monotonic in their use of the waste-time data, incorrect value assignments should be quickly servoed out.

Finally, the master program will presumably be in charge of whatever evolutionary processes are to be applied to the sets of Methods and Characters (and perhaps Models). Two remarks on this aspect.

1. As McCarthy has observed, it would be very nice if we could find a set of problems, plus a master program, which would work together in such a way that the solutions to the problems that the machine finds on the base level could be used by the master program to guide and improve the second level evolution or, as McCarthy would prefer, to directly improve itself.
2. On a less ambitious level, it seems to me that in choosing where to put in machine time on the second level evolutionary search process, one should start with a preference for working on the characters. The reason is that if the characters are good (in that they reflect relevant aspects of the problems)

then one can set up an evolutionary process for the Methods, without any unusual difficulty. On the other hand, if the characters are no good, then the Methods just can't evolve efficiently. Too much information is lost. For each problem, there might be highly appropriate methods available in the Method Box, yet the machine would have to select them essentially at random. I think that the situation is essentially unsymmetrical here, perhaps because of some "irreversibility of information loss" effect.