Limitations of the Current Stock of Ideas about Problem Solving*

ALLEN NEWELL

Institute Professor
Systems and Communication Sciences
Carnegie Institute of Technology

17

Limitations of the Current Stock of Ideas about Problem Solving*

ALLEN NEWELL

Institute Professor
Systems and Communication Sciences
Carnegie Institute of Technology

Figure 1 shows a checkerboard with a domino beside it. The domino covers exactly two squares of the board. Suppose we are given an unlimited supply of dominoes and asked to cover the checkerboard exactly—i.e., with no dominoes extending over the boundary. This is a trivial problem. The dominoes can be laid down as in Fig. 2; and there are many other arrangements that would do the job equally well.

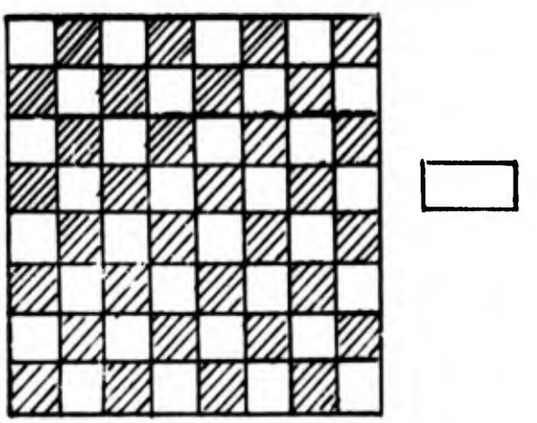

Figure 1. Checkerboard.

Now let us mutilate the board, as shown in Fig. 3, by removing the two corner squares. Again, the problem is to cover the board with dominoes. Only this time it is a hard problem. In fact, it is impossible. Therefore, the real problem is to prove that it is impossible. (Before reading further try to convince yourself of the impossibility and try to find a proof. You may already know the problem, of course, since it is a familiar chestnut.)

*The preparation of this paper was supported by Contract SD-146 from The Advanced Research Projects Agency of the Defense Department.

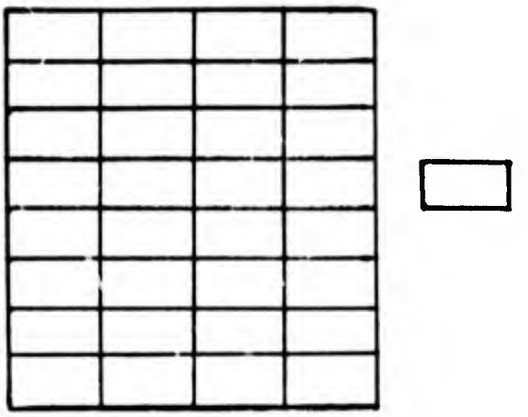

Figure 2. Covered checkerboard.

Most people find the proof difficult to discover, but transparent once found. Observe that the original checkerboard has thirty-two black squares and thirty-two white squares, and that a domino always covers one black square and one white square. With two white squares removed, the mutilated board has thirty-two black squares and only thirty white squares. Consequently, no matter how the dominoes are laid down eventually a position will occur with two black squares left and no white squares; and it will be impossible to cover these remaining two squares.

Our concern is with machines and not men. Hence, the ultimate problem is not to discover the proof, but to build a machine that can discover the proof to the domino problem. If is a fair statement, I believe, that no one today knows how to build such a machine—or equivalently, how to

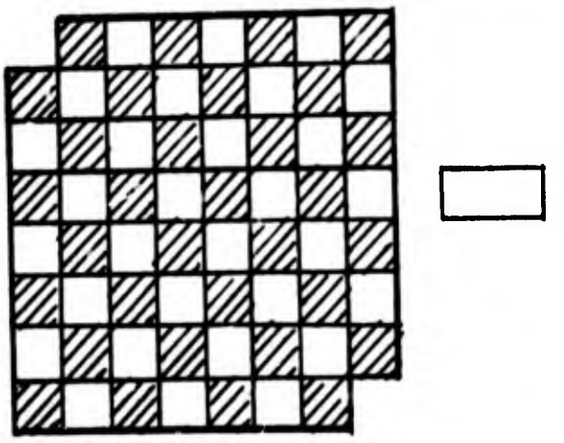

Figure 3. Mutilated checkerboard.

construct such a computer program. And this inability represents one of the limitations on the current stock of ideas about problem-solving by computers.

It may seem disturbing to have a limitation of ideas stated as the inability to solve a particular problem. It doesn't say what is missing. Even admitting that to say exactly what is missing is to say too much, one might still hope to describe classes of problems that could not be solved. Instead, the domino problem seems extremely particular.

In fact, proceeding by highly particular examples is characteristic of work in programming computers to solve problems. It is standard methodology—to write specific programs to do specific things—and in its own way represents a limitation on our current stock of ideas. Nevertheless, it is possible to use a single example as a tool to explore more generally our current knowledge about how to make computers into problem-solvers.

THE PROBLEM OF REPRESENTATION

The experience of many people with the domino problem is that they have no idea at all how to get started on finding a proof. When and if a proof is found, it occurs suddenly. This leaves them with a proof, but with no idea at all how a program might find it. Let me interpret this experience. Progress on a problem requires having some representation of the possible solutions to the problem that can be manipulated, searched, or explored in the process of determining the correct solution. With no representation, there is no possibility of manipulation and no way of making progress. Thus, the initial "lost" period is in fact devoted to finding a representation. The suddenness of solution arises from the extreme simplicity of the proof, so that once a representation is found, the "essential idea" of the proof is immediate, as is the verification of its soundness. Thus, there is little awareness of the representation of the possible proofs, which is what is needed to make a start on a computer program for finding the proof.

The proposition that a representation of possible solutions is necessary to finding a particular solution appears almost banal. However, the existing lines of attack in getting computers to problem-solve can be described in terms of the representations they have developed. And an important aspect of their limitations can be seen in what kinds of problems can be easily cast into these representations. We will put some flesh on this proposition by considering a number of these representations. As a common thread, we will ask whether each representation could help us in building a program that would find the proof of the domino problem.

HEURISTIC SEARCH

Perhaps the most notable approach in problem solving by computers is heuristic search. Almost all the successful theorem-proving, game-playing and puzzle-solving programs of the last several years belong in this class, as well as a number of programs for management problems of scheduling and allocation.[1] The basis of heuristic search is that I can look at any problem as if there are a set of situations (say S_1, S_2, ...) and a finite set of operators (say Q_1, Q_2, ... Q_n), such that given the situation S_i, the application of an operation, say Q, transforms the situation into another one, say S_j. As Fig. 4 shows, the situations can be viewed as the nodes of a tree, with the operations as the branches. The application of a sequence results in searching a part of the tree.

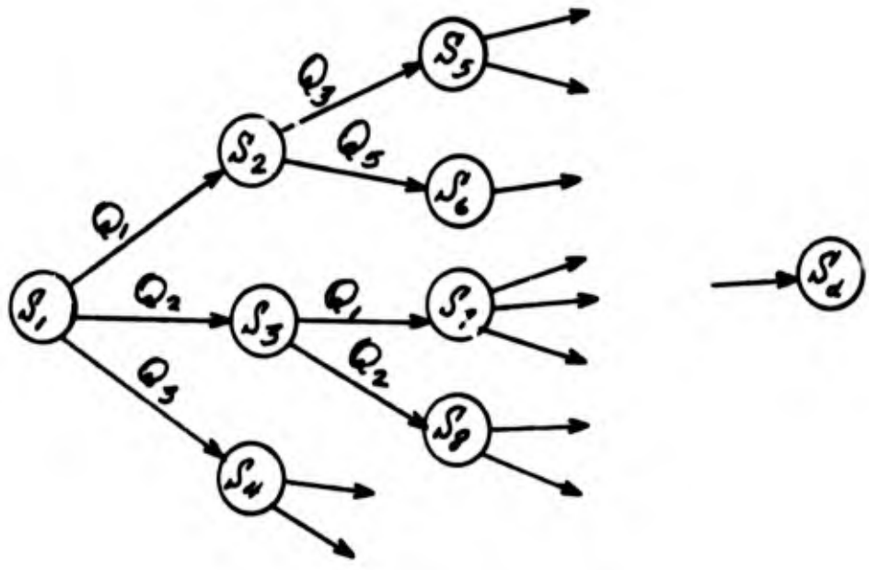

Figure 4. Tree representation of problem.

In this representation, a problem takes the following form: The initially given situation is the root of the tree, S_1; the desired situation is some S_d (possible a set of situations); the problem is to obtain S_d starting from S_1. Thus the problem is one of searching through the tree (as implied by applying different sequences of operators) until S_d is discovered. To pick one concrete example, if the game is checkers, the situations are checker positions, the operators are the legal checker moves, S_1 is the opening position, and the desired positions are those in which your side wins.

When problems of realistic difficulty (like chess and checkers) are cast into this representation, the trees turn out to be massively large and the problems cannot be solved simply by searching at high speed. Instead various rules (called heuristics) are used to narrow the search to the profitable part of the tree. These rules can be evaluation functions on the situations that approximate the final value, or rules that eliminate a branch, or rules that determine how much effort should be spent in searching a subpart of the tree. We are not interested here in the particular form of these heuristics. What is of interest is that having once represented the problem as search in a tree, there are a number of things we can do to bring the computer's problem-solving power (here, its capacity for sophisticated search) to where it solves significant problems. Indeed, the computer itself can modify and extend its own heuristics. For example, Samuel's checker program[4] modifies its evaluation function on the basis of its past experience.

Let us return to the domino problem and ask whether these ideas are of use. We can certainly represent the domino problem itself in this way: the situations are all the partially covered checker boards; the operators are the placing of a domino either vertically or horizontally so it covers two squares not yet covered; the initial situation is the empty checkerboard; and the desired situation is the completely covered board. But this doesn't lead anywhere. If coverings existed, a program could find them this way. But if coverings are impossible and the job is to prove it, then trying possible coverings, no matter how many, doesn't help a bit. Only if the program tried all possible coverings and knew it had exhausted them could it conclude that none were possible. But this implies searching the entire tree, and the tree is much too big (at least 10^{20} situations).

PREDICTING SEQUENCES

Let us turn to a different task, which has been solved by programs of a somewhat different kind.[6] The problem is to predict the next letter in the following sequences:

1. A B A B A B ___
2. A T B A T A A T B A T ___
3. D E F G E F G H F G H I ___

The answer to the first is clearly A; the answers to the others are not quite so clear, but are attained without difficulty by intelligent humans. However, for us the problem is not how humans can do it, but how to write a computer program that will do it.

This seems a difficult task—indeed, it involves a genuine induction—until one notes the absence of a representation of possible patterns, and takes steps to provide it. Consider the following scheme, which we can illustrate on the second task. A sequence will be generated by the iterated application of a set of rules; this set of rules, therefore, represents the pattern. There will also be some variables that maintain a memory of the current cycle, upon which the rules can act. For the second pattern, we start with one variable, m_1, which takes values in the alphabet (A, B) and initially has the value *B*. The rules are given by the expression:

$$A, T, m_1, n(m_1)$$

This is to be interpreted: Print A; then print T; then print the current value of m_1; then change the value of m_1 to be the next higher letter in the alphabet of m_1. Thus, on the initial run this prints ATB and changes m_1 to A (the alphabets are understood to be cyclical). The next run yields ATA and m_1 changes to B, and so on.

To give one more example, the third sequence above requires two variables, m_1 and m_2, both of which range over the standard alphabet (A, . . . , Z) and have initial values of *D*. The iterative rule is given by the expression:

$$m_1, n(m_1), m_1, n(m_1), m_1, n(m_1), m_1, n(m_2), e(m_1, m_2)$$

The first seven steps of this expression generate the four letters in a cycle; e.g., DEFG. Then m_2 is advanced one (e.g., from D to E) and m_1 is set equal to it. Thus the next cycle goes EFGH.

Once this language of patterns has been defined it is easy to write a program that will interpret it; that is, that will generate the sequence, given the expression. More important, it is also easy to construct a program that will discover whether any simple expression in this language agrees with a sample of a sequence. Given the language, it is clear that one must conjecture the cycle in the sample, and then discover the relations (expressed in terms of the operators *n*, *e*, and the various alphabets) between the letters both within the cycle and between corresponding members of successive cycles. Partial solutions can be tried (via the interpreter) and the discrepancies used to modify the expression. In short, once a representation is available for possible solutions, it is possible to construct programs that work on the problem in reasonable ways.

Returning to the domino problem, it hardly seems possible to apply the above language directly. Rather, we should look to the principle involved: "Build a language to express the possible solutions." Our problem is to find a language of proofs. We already have a way of talking about

checkerboards and various coverings of dominoes; this clearly is not enough. Since proofs are normally given in a combination of natural language and notation about the task (this latter corresponding to our checkerboard and coverings) it is not easy to imagine what such a language of proofs might be like. However, there has been considerable work in constructing computer programs to find proofs, and we can look at these.

THEOREM-PROVING IN THE PREDICATE CALCULUS

Currently there are two distinct approaches to theorem proving. One of these considers the problem as one of heuristic search. The situations are theorems, the operators are the rules of inference, the initial situation is the collection of theorems that can be assumed true, and the object of search is the desired theorem. This approach has worked in areas where the rules of inference and the possible theorems are clearly set out, as in plane geometry or symbolic logic. But in the domino problem our difficulty is that we do not have any language for expressing possible theorems (other than the one given), nor are the rules of inference delineated. So we must solve our problem of representation prior to using heuristic search techniques for discovering the proof.

The second approach appears more hopeful. The development of mathematical logic has resulted in some formalized logical systems of great scope and power. One of these, called the first order predicate calculus, has received a great deal of attention from logicians interested in constructing programs to prove theorems. This calculus permits assertions involving the usual logical connectives (and, or, not, implies) and in addition, assertions of the form "There exists an x such that $A(x)$ is true" and "For all x, $A(X)$ is true," where $A(x)$ is any legal assertion in the calculus and x is a variable ranging over the basic objects that the calculus makes assertions about. The appeal of this system is not just that a great deal is understood about it mathematically, but that it appears to be rich enough in expressive power to cover most of the mathematics used in science and engineering. This gives rise to a vision in which all problems of proof are translated into the first order predicate calculus, and a single big theorem-proving engine is built for handling proofs in this calculus. Thus, the predicate calculus provides a universal means of representation. This vision has sufficient appeal that an entire subfield of artificial intelligence is devoted to its implementation, and numerous programs have been built to prove theorems in this system.[8]

Certainly we should apply this to the domino problem. First, we must

translate the problem into the predicate calculus; then we can explore the possibility of current programs proving the theorem. Of course, there is more than one way to represent the domino problem in the predicate calculus—so the task of translation should not be passed over too lightly. However, analogously to the sequence-predicting problem already discussed, a representation already exists so the problem is quite tractable. We will not provide a translation here; it is too technical for this paper. Recently, however, John McCarthy has published a short memo, entitled, "A Tough Nut for Proof Procedures,"[3] in which he provides a translation of the domino problems into the predicate calculus and asserts that this theorem will be very difficult for present theorem-proving programs to handle. To quote him, "... I don't see how the parity and counting argument can be translated into a guide to the method of semantic tableaus, into a resolvant argument, or into a standard proof. Therefore, I offer the problem of proving the following sentences inconsistent as a challenge to the programmers of proof procedures and to the optimists who believe that by formulating number theory in predicate calculus and devising efficient proof procedures for predicate calculus, significant mathematical theorems can be proved." ["Semantic tableaus" and "resolvant arguments" are two special techniques developed in the field. "Proving the ... sentences inconsistent" refers to a standard approach in the field of conjoining the axioms and given theorems with the *negation* of the desired theorem to obtain a contradiction.]

PATTERN RECOGNITION

Let us consider just one more class of tasks, that of recognizing a pattern. Typical of such problems is recognizing the letters of the alphabet when printed, or when written by hand. Many computer programs (and hardware devices) have been constructed that do moderately well at these tasks; harder tasks are recognizing spoken words, or human faces. Now, an important superficial characteristic of human pattern recognition is that it appears to occur "all at once"—immediately, without protracted inferences. This is reminiscent of the suddenness with which most people discover the domino proof—"nothing" for a while, and then the proof is simply "there." Thus, we might look at pattern-recognition programs to see how they represent problems and whether this representation might be of use with the domino problem.

Enough pattern-recognition programs have been constructed, so we have a pretty good idea of the basic components. (At least, those that have been built have much in common; there might be other approaches which no one has discovered yet.) As Fig. 5 shows, there is an initial com-

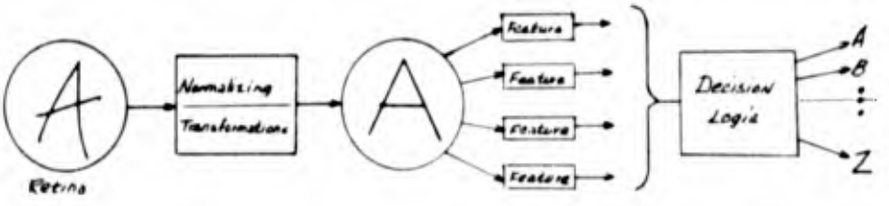

Figure 5. Schematic pattern recognizer.

ponent in which the item to be recognized is registered, often called the retina for obvious reasons. Then occurs a series of normalizing transformations, which get rid of variation by putting the input into standard form. In visual recognition these are such operations as centering, focusing, smoothing, enhancing contrast at edges, etc. Following this there are a set of feature detectors; each one reacts to some characteristic of the image. Taking vision, again, these might be "the existence of a vertical line segment," or "the number of corners," or "a marbled texture." Some of these features are themselves moderately complex, and may be thought of as involving the combination of other features. Finally, there is a component that combines all these features and arrives at a decision. This might be a "decision tree" in which discriminations on the various features finally lead to identifying the pattern; or it might involve measuring how close the input image is to templates of the possible patterns and choosing the closest.

The scheme of Fig. 5 can be taken as another general representation of how to make decisions or selections. Given a new task, the scheme directs attention to what pieces need to be defined and how they should then be related to produce a total system. It does not provide a representation of the possible solutions; rather, it is a representation of the problem-solving process. This is unfortunate, since if we try to apply the scheme to our domino problem, it provides us with little clue as to what should be made available at the retina (surely the checkerboard, but what else?), what features should be taken, or what the class of responses should be from which the right one (the proof) should be selected.

Although not appearing to help directly with the domino problem, the area of pattern recognition provides a good historical example of the dif-

ference between having a representation and not having one. In visual or auditory recognition, the representation on the retina and the set of responses are quite well defined; the real questions focus on the transformations, the features, and the decision logic. Of these, the features have seemed especially critical. A few years ago, it was an informal maxim in the field that one could undoubtedly design, ad hoc, a good set of features for any specific limited recognition task, but that the "real problem" was how to get new features for new tasks.[5] Up to this time, the features had always been thought up by the programmer on the basis of prior experience and investigation and simply programmed into the recognition program. The features that worked for one task did not necessarily work for another. The inability to construct recognition programs which built their own features was considered a significant limitation of the field.

In 1961 Leonard Uhr developed the first successful pattern-recognition program that obtained its own features.[7] The details of this program are not of interest here, but the essential idea is important. Since features had been anything a programmer could think up (as, for us, are the ideas for proving the domino theorem), there was no way of talking about the set of possible features (nor, for us, the possible proofs). Hence, there was no way of getting a program to manipulate features and develop new ones. Uhr's main contribution was to construct a space of possible features. The retina in his program was a rectangular grid of bits, 20 on a side, as in Fig. 6. The pattern to be recognized is written on the blank retina (consisting of all 0's) by putting 1's in the appropriate cells. A feature, said Uhr, is defined by a 5 × 5 subgrid having 0's, 1's and *X*'s for entries (only the *X*'s show in Fig. 6 to avoid confusion). The subgrid is swept over the entire matrix; at each position a measure of agreement be-

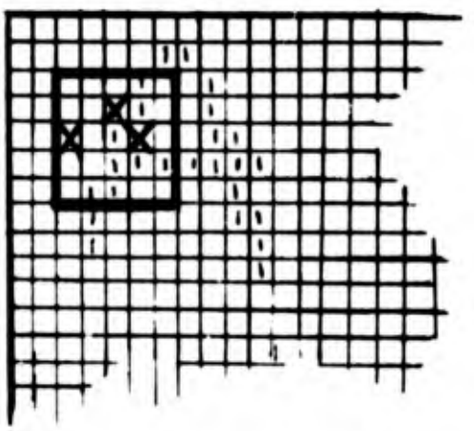

Figure 6. Retina and 5 × 5 feature.

tween it and the retina is taken by counting the 0's and 1's that match (and ignoring the X's). This distribution of measures is used to define the actual feature—e.g., the position where it is strongest, whether it ever exceeds a certain threshold, etc. The important thing for us is that there exists a set of possible features (all different subgrids), so that the program could introduce new ones. For instance, it could copy a part of a sample pattern and use it as a feature in recognizing other exemplars of the same pattern. This is an extremely simple scheme, almost naive; yet it was enough to permit his program to recognize a wide variety of different kinds of patterns, developing for each an appropriate set of features. It was enough to dispose of the maxim.

A FINAL LOOK AT THE DOMINO PROBLEM

Although the domino problem is not easily assimilated to any existing approaches, each of them has had something to say about how to represent a problem and how to proceed to solve it. Together they permit a slight reformulation of the domino problem. This is of interest in showing that, having represented the problem and surmounted one hurdle, the next hurdle we come to is again a matter of representation.

As noted earlier, we can formulate the task of covering the checkerboard as a tree of operations. Clearly, we can get the computer to try a series of domino placements, Q_1, Q_2, . . . , starting at S_1 to attempt to get a complete covering (see Fig. 4 again). Since the task is impossible, there is no path that leads to S_d, the final, perfect covering.

Now there must be something that prevents a path starting at S_1 from reaching S_d. That is, there must be some property of the initial situation that is true of all the situations (the S_i) reachable from S_1, is not true of S_d, and such that none of the operators, Q, changes it. Putting this more formally, let $P(S)$ be this property, determinable for any position. Then the conditions are:

1. $P(S_i)$ is true.
2. If $P(S_i)$ is true then $P(Q(S_i))$ is true for any legal Q.
3. $P(S_d)$ is false.

And the conclusion is:

There is no sequence $Q_1, Q_2 \ldots Q_m$ such that

$$Q_m Q_{m-1} \cdots Q_2 Q_1(S_1) = S_d$$

Proposition (1) says that the property is true of the initial situation. Proposition (2) asserts that this property is hereditary; that is, if it holds for a situation, it holds for all those that immediately follow from it by legal moves—hence, for any that can be reached through any chain of legal moves. Finally, proposition (3) says the property does not hold for the desired position. The conclusion is that the final position can never be reached.

Note that the actual proof can be put in just this form. The property P is that the number of black and white squares uncovered are unequal. This is true of the initial board; and the placing of any domino, which covers one square of each color, leaves the property true of the resulting board. But the final position has equal numbers uncovered, namely, zero.

If the problem is reformulated as above, then the task shifts to the search for a property with the desired characteristics. But first it is necessary to ask whether a computer program could be expected to reformulate the task in this way. This seems reasonable to me, in support of which I offer the following plausibility argument. The formulation above is an example of the principle of mathematical induction, usually stated, "If $P(n)$ implies $P(n + 1)$, and $P(1)$ is true, then $P(n)$ is true for all positive n." Now there is only one such principle, just as there is only one concept of equality, one concept of a function, or one mathematics of the integers. Consequently, it is reasonable to assume that a problem-solving program would be given this principle. In fact, this is the way almost all humans get their basic intellectual tools. (That they are not easily discovered by the unaided human intellect is testified to by the long historical development of mathematics.) Therefore, the program does not have to discover the induction principle; it has only to evoke it and apply it. To evoke the principle does involve a recognition; however, there are relatively few basic ideas for proofs, so that this is not the difficult step. Likewise, transformation of the principle from its positive form into the essentially negative form used in the domino proof does not seem insurmountable. The machine has a representation of the principle and a representation of the final thing it wants to prove—i.e., proposition (4). Purely formal operations can be used to manipulate the principle to give (1), (2) and (3).

Despite the unfilled gaps—several programs have been built to use the principle of induction in sophisticated ways,[2] but none to adapt the principle to new situations—let us accept that the program can get as far as the formulation (1)-(4). Where does it go from here? Its task is now to find a feature. Again, the difficulty is that no space of features is given within which to search—i.e., a representation is missing. If we limit the features too severely—e.g., to relations among numbers of black and white

squares, then in choosing the space of features we have already done most of the work. That is, it is we who have found the proof by selecting the feature space. If, on the other hand, we give it no representation at all, then the program can do nothing. It is not enough to give it the checkerboard; it must also have ways to measure aspects of the board and to combine and compare these in various ways. Even at this stage, for instance, it is clear that it makes a great deal of difference whether the program is given a checkerboard, with its squares alternating in color in the relevant way, or whether it is given a blank board. (Only the checkerboard's familiarity inhibits the checkering from immediately cluing the human.)

Actually matters are not quite so difficult, since the expressions (1)-(4) provide some good raw material to work with. However, in the interests of making the point we will not press the example to the limit. (For I believe, certainly, that given a modest amount of additional effort, a reasonable program can be constructed that finds the domino proof and does so fairly.) It is enough to observe the transformation of the original problem of representation into another (less severe) problem of representation.

CONCLUSION

Let me summarize the general argument, for which the domino problem has been only a means, although hopefully an entertaining one. We can look at the current field of problem solving by computers as consisting of a series of ideas about how to represent a problem. If a problem can be cast into one of these representations in a natural way, then it is possible to manipulate it and stand some chance of solving it. Different approaches, consisting of different global visions about representation, are not easily translatable, one into the other. Naturally, each of these visions turns out to have certain advantages and certain disadvantages, much of which can be summarized by describing the kinds of problems which can be easily so represented, and admitting that we can't yet stretch any one representation too far.

The natural response to this description of problem solving is to inquire where representations come from, and what is known about constructing new ones. Here we are on familiar, but unpleasant, ground. Currently, representations seem to arise in isolation—"out of nowhere." To put it in still more familiar terms, we do not yet have any useful representation of possible representations. This is possibly the biggest limitation on the current stock of ideas about problem solving.

REFERENCES

1. Feigenbaum, E. and J. Feldman (eds.), *Computers and Thought* (McGraw-Hill, 1963). Contains many examples, reprinted from the primary literature.

2. London, R., "A Computer Program for Discovering and Proving Recognition Rules for Backus Normal Form Grammars," *Proc. Assoc. for Computing Machinery*, A1.3-1–A1.3-7 (1964), p. 64.
3. McCarthy, J., "A Tough Nut for Proof Procedures," Stanford Artificial Intelligence Project Memo 16, July 17, 1964.
4. Samuel, A., "Some Studies in Machine Learning Using the Game of Checkers," *IBM J. Research and Development*, vol. 3 (July 1959), pp. 211–229. (Also reprinted in Feigenbaum and Feldman.)
5. Selfridge, O. G. and U. Neisser, "Pattern Recognition by Machine," *Scientific American* (August, 1960), pp. 60–68. See especially the last paragraph. (Also reprinted in Feigenbaum and Feldman.)
6. Simon, H. A. and K. Kotovsky, "Human Acquisition of Concepts for Sequential Patterns," *Psychol. Rev.*, vol. 70 (November 1963), pp. 534–546.
7. Uhr, L. and C. Vossler, "A Pattern Recognition Program That Generates, Evaluates, and Adjusts Its Own Operators," *Proceedings of the Western Joint Computer Conference*, vol. 19 (1961), pp. 555–570. (Also reprinted in Feigenbaum and Feldman).
8. Wos, L., D. Carson, and G. Robinson, "The Unit Preference Strategy in Theorem Proving," *Proceedings of the Fall Joint Computer Conference*, vol. 26 (Spartan, 1964), pp. 615–621.