8

A Theory of Advice

Donald Michie

Machine Intelligence Research Unit
University of Edinburgh

Machine intelligence problems are sometimes defined as those problems which

(i) computers can't yet do, and
(ii) humans can.

In a try for a less *ad hoc* formulation we shall say that a machine intelligence problem is one whose solution program

(i) is time-infeasible if minimally represented, but
(ii) can be made time-feasible by a feasible memory extension containing "advice."

We shall further consider how much "knowledge" about a finite mathematical function can, on certain assumptions, be credited to a computer program. Although our approach is quite general, we are really only interested in programs which evaluate "semi-hard" functions, believing that the evaluation of such functions constitutes the defining aspiration of machine intelligence work. If a function is less hard than "semi-hard," then we can evaluate it by pure algorithm (trading space for time) or by pure look-up (making the opposite trade), with no need to talk of knowledge, advice, machine intelligence, or any of those things. We call such problems "standard." If however the function *is* "semi-hard," then we will be driven to construct some form of artful compromise between the two representations: without such a compromise the function will not be evaluable within practical resource limits. If the function is harder than "semi-hard," i.e. is actually "hard," then no amount of compromise can ever make feasible its evaluation by any terrestrial device.

"Hard" problems

In a recent lecture Knuth (1976) called attention to the notion of a "hard" problem as one for which solutions are computable in the theoretical sense but

not in any practical sense. For illustration he referred to the task, studied by Meyer and Stockmeyer, of determining the truth-values of statements about whole numbers expressed in a restricted logical symbolism, for example

$$\forall x\, \forall y(y \geqslant x+2 \Rightarrow \exists z(x<z \wedge z<y)), \text{ or}$$

$$\forall S(\exists x(x \epsilon S) \Rightarrow \exists y(y \epsilon S \wedge \forall z(z \epsilon S \Rightarrow y \leqslant z)))$$

Thanks to a theorem of Büchi it is known that a device could in principle be constructed capable of evaluating the truth or falsehood of any valid input expression in this symbolism in a finite number of steps. But is the problem nevertheless in some important sense "hard?"

Meyer and Stockmeyer showed that if we allow input expressions to be as long as only 617 symbols then the answer is "yes," reckoning "hardness" as follows: find an evaluation algorithm expressed as an electrical network of gates and registers such as to minimise the number of components; if this number exceeds the number of elementary particles in the observable Universe (say, 10^{125}), then the problem is "hard." A consequence follows that any representation of the same algorithm as a computer program for any sequential machine *either* would entail for some inputs too many computational steps for solution within feasible time, *or* would contain too many symbols to be accommodated in any feasible store, *or* both. In drawing my attention to this consequence my colleague David Plaisted related it to recent combinational complexity studies *(see* Schnorr, 1975, Fischer, 1975).

Our definition of "semi-hard" adopts computer programs as the sole representational form for solution algorithms, and explicitly recognises *both* categories of resource-bound, namely time (e.g. number of computational steps) and space (number of store-bits). For each given finite function we consider two different representations:

(1) *space-minimal*: the shortest program which will evaluate the function for all inputs, no concern being given to the number of computational steps required, nor to possible work-space requirements during the course of a computation.

(2) *time-minimal*: the program which requires the least number of steps for worst-case evaluation, no concern being given to the program's length or work-space.

Questions of differential weighting for different kinds of computational step are neglected. Such questions are related to a valid objection to the effect that criteria (1) and (2) cannot anyway be applied without a complete specification of the machines on which the programs are to run. In a recent outline of work in algorithmic information theory, Chaitin (1975) considers this objection in an essentially similar context and dismisses it as quantitatively unimportant. The real point here is that problems in which we are likely to be interested typically exceed our arbitrary boundary lines by such large margins that the arbitrariness hardly matters. Differences introduced by envisaging one rather than another machine specification are as irrelevant to the main thrust as would be a decision

in the earlier example to credit the Universe with 10^{120} or 10^{130} elementary particles instead of 10^{125}.

Here are some boundary lines, scaled down for coziness from Universal to planetary dimensions.

A "semi-hard" problem is not "hard"; yet any space-minimal solution-program written for any envisageable sequential machine would for some inputs require a running time greater than the age of the Earth, and any time-minimal solution-program written for any envisageable sequential machine would require a store too large to be accomodated on the Earth's surface.

But surely a problem as hard as *that* must be "hard" in the full sense of Knuth, that it "will never be solved in our lifetime, regardless of how clever people become or how many resources are committed to the project?" Not so! Here indeed is the whole joy of the phenomena of human cognition and of cognitive engineering, that a loop-hole for compromise can exist between the two criteria of minimality. How do we know, for a given "semi-hard" problem, that we cannot devise a representation which, although not minimal, is still acceptably short *and* has a non-minimal but acceptable running time? Let us do some calculations around this thought, taking our "semi-hard" function from the game of chess.

A function f links each legal chess position to its game-theoretic value; that is, it maps onto the set $\{1, 0, -1\}$ corresponding to outcomes which are "won," "drawn" or "lost" from the standpoint of the opening player. What might be a space-minimal representation of this particular f? Strictly speaking we do not know. But we can show that such a representation must be very short, for an upper bound to the minimal length is given by the iterated minimax algorithm for computing f by (i) looking ahead along all possible continuation paths from the given position, (ii) using the rules of chess to assign outcome values to all the terminal positions of the lookahead tree, and then (iii) backing these values up the game-tree using the minimax rule until the root position has been labelled. A minimum-length implementation of this as a program requires of the order of 10^4 bits only. Hence *either*

A. 10^4 bits does indeed measure the space-minimality of chess ("complexity" in Chaitin's terminology, "ω-complexity" in ours (Michie, 1976)), *or*

B. f's space-minimal representation is even shorter than this.

Case B keeps alive the possibility that chess is not even "semi-hard," since this hypothetical shortest representation might turn out to be so fast-running as to evade the time-bound horn of the dilemma. This would be the case if some sweeping mathematization of chess were discovered, analogous to the parity rule for the game of Nim.

Case A, which says that we are not going to get a shorter rule than iterated minimax, corresponds to many people's intuitions. We shall assume it here for purposes of expounding the "semi-hard" idea. Note that we are not turning aside from the possibility of a compact mathematical rule for chess, but only from the more remote possibility that such a rule might be shorter even than iterated minimax.

We first re-state our essential position.

A "semi-hard" problem is a non-"hard" problem whose space-minimal solution exceeds practical bounds of time and whose time-minimal solution exceeds practical bounds of space.

Assuming Case A, chess meets the first half of the definition. The expected waiting time for evaluating f, according to a calculation of Shannon's (1950), would exceed 10^{90} years on a machine able to calculate one variation per micro-micro-second. Let us turn then to f's time-minimal representation.

Restricting ourselves, as we have deliberately done, to computer programs for sequential machines, we adopt the look-up table as the time-minimal representation. We make any necessary assumptions concerning the relative speeds of look-up operations *versus* other operations so as to ensure that this representation comes out time-minimal in all cases. Determining the space requirement is then equivalent (disregarding the space-occupancy of the look-up program itself) to calculating the information-content of a message which encodes the extensional form of f, i.e. the sequence of symbols taken from f's co-domain Y corresponding with an ordering of the set X of size N which is f's domain. This information-content, $I(f) = -N\sum_{Y} p(y)\log_2 p(y)$, tells us the length of the shortest such message and hence the theoretical minimum number of store-bits required to hold the look-up table. The choice of ordering for the domain is treated as arbitrary, in the sense that the encoding may not exploit it to achieve additional compression, although of course free to exploit the first-order frequencies of the different y-values in the message.

On this basis the look-up table for f would require something between 10^{45} and 10^{50} bits of store, corresponding roughly to the range of the various estimates which have been made of the number of different legal chess positions. A store of this size, for any conceivable advance in micro-miniaturization, could not be assembled on the surface of the Earth. Chess, then, even in the restricted sense of recognising the game-theoretic value of a position, qualifies as at least "semi-hard."

At this point the task of evaluating f may seem a matter for despair. But all that we have so far indicated for chess is that the space-minimal solution may not be time-feasible and that the time-minimal solution is not space-feasible; *not* that no solution-program of any kind could combine the two feasibilities. Only this last case corresponds to the (unproved) hypothesis that chess is "hard."

To clarify the definition of feasibility, and to recapitulate the ideas so far discussed, Figure 1 presents "trade-off curves" for a series of hypothetical functions: two "semi-hard" functions and a "hard" function. If position-evaluation in chess is like f then error-free performance at least under correspondence chess conditions is feasible. If it is like g, then any given position can be evaluated, but at the expense in the worst case of a year's continuous running-time using a 10^{15}-bit random-access store. If it is like h, then no amount of time and memory-space within terrestrial limits would be sufficient.

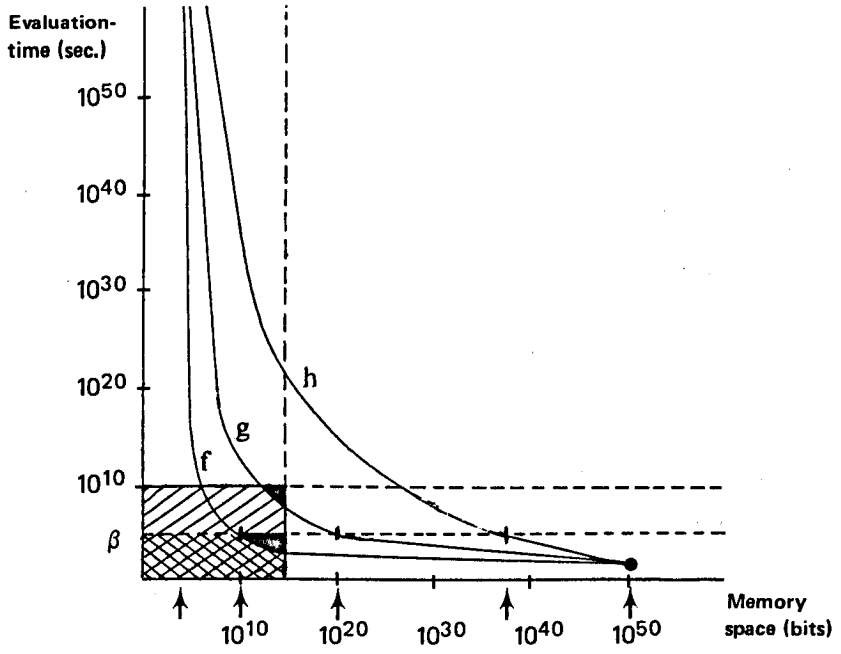

FIG. 1. Store-time trade-off curves for hypothetical finite functions f, g and h. Each has the same information-content (10^{50} bits) and the same ω-complexity (10^4 bits). Time-feasibility and space-feasibility limits have somewhat arbitrarily been placed at 10^{10} secs. and 10^{15} bits respectively. Time-acceptability is set at 10^5 secs, indicated by β. The hatched rectangle is the "zone of feasibility," through which curves f and g pass. Only f passes through the cross-hatched time-acceptability sub-zone. The five upward arrows mark respectively: the ω-complexity of f, g and h; the β-complexity of f; the β-complexity of g; the β-complexity of h; the information-content of f, g and h. f is "semi-hard," and since its β-complexity is less than the space-feasibility bound it is "not too difficult." g is "semi-hard" and "too difficult." h is "hard," which renders it *a fortiori* "too difficult."

Figure 2 depicts a special kind of "hard" function and reveals a twist in our formulation. In his paper Gregory Chaitin discusses the function which maps from major-league baseball games to their scores. "In this case" he remarks "it is most unlikely that a formula could be found for compressing the information into a short message (contrast the iterated-minimax formula for chess–D.M.); in such a series of numbers each digit is essentially an independent item of information, and it cannot be predicted from its neighbours or from some underlying rule. There is no alternative to transmitting the entire list of scores." Now suppose that there are so many major-league results (10^{50} if you like; never mind the time taken to accumulate them!) that any solution program for this function, even though time-feasible, would necessarily be space-infeasible, consisting indeed merely of a giant look-up table. The function, if we did not already know it to be "hard," would then fail to satisfy the first limb of our definition which says that a "semi-hard" problem's space-minimal solution is time-infeasible. As we progressively scale down the hardness of a problem of this

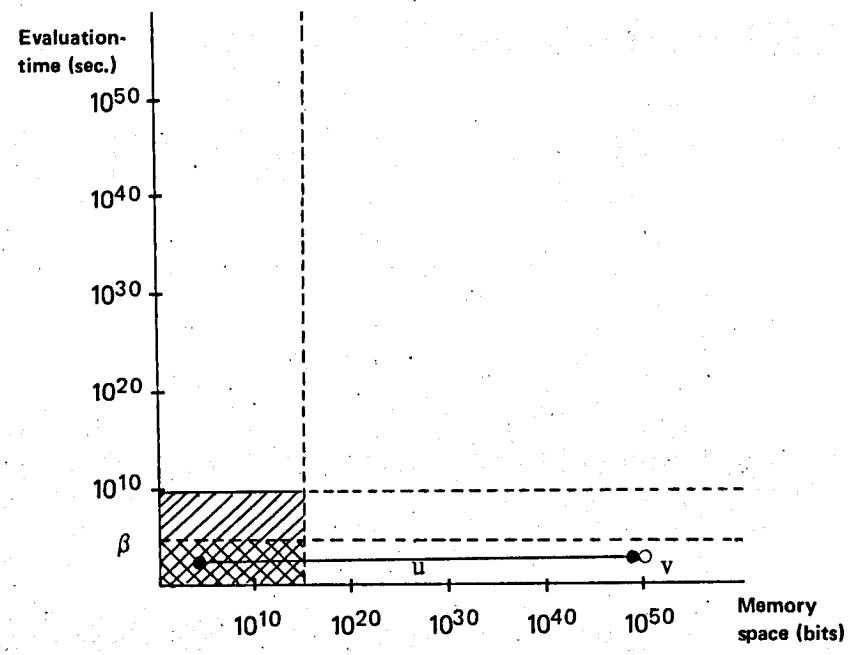

FIG. 2. Function v (open circle in the diagram) is "hard," and it cannot be compacted. Like Chaitin's "baseball" function it is devoid of internal structure and hence offers no trade-offs. Function u is of an opposite type. Its representation can be compressed into a highly compact solution program which (in contrast to iterated minimax in chess) is not appreciably slower-running than look-up. An example might be position-evaluation in games of Nim starting with, say, 17 heaps of 1000 pieces each.

type by gradually decreasing the domain size, the transition from "hard" to "standard" occurs abruptly without passing through a zone of "semi-hardness." Possessing no exploitable structure, such problems offer no hand-holds to machine (or any other) intelligence. The Figure also shows, for contrast, a "standard" problem of the super-exploitable "Nim" type.

Our definition may not seem entirely satisfactory, since it catches in its net problems which we have come to regard informally as not so difficult, for example sorting a list. It may well be that the space-minimal solution of the sorting problem is none other than the simpleton program which repeatedly traverses the whole list, swapping neighbour pairs whenever the sort relation shows them to be the wrong way round—time-infeasible for quite a modest list size. Hence even though numerous fast methods are known today which would make light work of it, sorting such a list shows up as "semi-hard" on our definition. So be it. We often forget how difficult a problem really is once it has succumbed to an intensive search for good solution methods (see also the recent appearance of sorting as a domain for machine intelligence work, as in Barstow and Green, this volume).

We now turn to "advice," seen as an approach to the solution of semi-hard problems.

Advice

Suppose that we have a space-minimal program, or indeed any short naive program which is time-infeasible, and we want to make it feasible. Two contrasted paths offer themselves, the path of the Great Leap Forward and the path of Incremental Advice.

Great Leap Forward. We scrap our naive program. Then after profound analysis of the problem we write a somewhat longer program expressing a fundamentally new algorithm. Here are three examples.

1. Sorting large lists. A naive program sorts by next-neighbour swapping. Hoare then creates Quicksort.

2. Factorizing large integers (like $2^{128} + 1$). A naive program counts up the number series doing divisions, first into the input number, then into the divisor and quotient found by successful division and so on. Solution times for a superfast machine would be measured in thousands of years. Brillhart and Morrison (cited by Knuth) devise a program to do it in an hour or two "by a combination of sophisticated methods, representing a culmination of mathematical developments which began about 160 years earlier."

3. Chess. A naive program executes the iterated-minimax rule, with a running time in the region of 10^{90} years. Some genius yet unborn devises a formula for identifying positions according to their game-theoretic value. Let us suppose that the new formula, unlike the Nim rule, is bulkier to represent than iterated minimax, but that it is nonetheless reasonably compact, and time-feasible.

The Great Leap Forward is without doubt the path of honour. Possibly progress towards solution of most "semi-hard" problems can and should be made in this way. Sometimes, however, we cannot wait for someone to make the Leap; or quite simply we are impressed by the fact that human expert performers in the given domain compute quite good solutions (as in chess) by methods which give no evidence at all of the kind of unitary insight associated with a Great Leap Forward. Also, there is the risk that we might have to wait forever, since some of the intellectual skills of man may owe their mosaic quality not to the limitations of their possessors but to lack of any deep structure in the given problem-domain for mathematical insight to seize upon.

Incremental Advice. We retain our naive program essentially unaltered, but from time to time we add to the store new materials which do not of themselves perform any computations necessary to evaluation but which act solely to expedite the operations of the naive evaluation program. This catalytic material is "advice." To illustrate we now consider the prime-counting function, which, given an integer, returns the number of primes less than that integer. The evaluation of this function is not the kind of problem normally associated with machine intelligence work. But Watterberg and Segre have analysed this straightforward and classical numerical problem in such a way as to display the Incremental Advice idea in a peculiarly simple and compelling fashion. They also nicely demonstrated a special advantage of the "advice" approach, namely that it directly lends itself not only to incremental additions by the user but also by

the program itself ("learning"). The following account is excerpted from their own report of the work (Watterberg and Segre, 1976).

> The work was done using a Digital Equipment Corporation PDP-11/35 running the UNIX timesharing operating system. All the programs were written in either PDP-11/35 assembly language or the UNIX system language, "C." Due to the nature of the UNIX timesharing system, the machine specification, S, on which the test program were executed, is simply a PDP-11/35 with 26K bytes of core memory. For information on instruction times and core cycle times, see the PDP-11/35 reference manual and the PDP-11/35 peripherals handbook.
>
> Three different function evaluation programs were written. Each had a different level of advice and/or learning ability. A short description of each follows.
>
> Program 1. Pure algorithmic
>
> In order to evaluate the function this program considers each integer from 2 to the function argument (P) minus 1 as a prime-candidate. Each candidate is tested for "primeness" by a division algorithm with the integers from 2 to sqrt(P) as the divisors. The prime-testing function is iterative and has no memory of any previous calls to it.
>
> Program 2. Advice
>
> Identical to Program 1 with advice added. The advice provided is an extension of the obvious fact that (apart from 2) even integers are not prime. If the multiples of 3 and the multiples of 5 are also removed from consideration, 8 integers in 30 remain as prime candidates as follows:
>
> Any integer can be expressed as
>
> $$30n + k, \quad 0 \leqslant k < 30.$$
>
> Since 30 is divisible by 2, 3, and 5, the only integers that might be prime are those for which k is not divisible by 2, 3, or 5, i.e. k = 1, 7, 11, 13, 17, 19, 23, 29.
>
> This advice was implemented as a gap table (consisting of the 8 gaps) which kept track of how much to add to the last prime-candidate to obtain the next prime-candidate. Note that this advice is also of use in the prime-testing routine since only those 8 in 30 numbers need be used as divisors.
>
> Program 3. Advice and Rote Learning
>
> Identical to Program 2 with a rote dictionary added. The rote dictionary contains ordered pairs of integers remembered from previous calls to the program. The rote dictionary is consulted before computation begins to find the largest ordered pair less than P. This value of the co-domain is used as a starting point for further compu-

	β	Program 1	Program 2	Program 3 (25 trials)	Program 3 (250 trials)
X	10 secs.	2050	6 560	74,213	399,487
	20 secs.	3630	11,160	125,201	676,589
	30 secs.	5100	15,200	166,159	903,589
L		5.15×10^3	7.0×10^3	12.1×10^3	26.5×10^3

TABLE 1. Data from Watterberg and Segre's programs 1, 2, and 3. X is the largest x for which the program could evaluate the function within β seconds. L is the store-occupancy of the given program. The last two columns relate to "learning" periods of 25 and 250 trials respectively. These experiments were run with a helpful tutor, i.e. the questions were put to the program in an order which maximised its learning rate.

> tation. A more efficient scheme would have required one half the rote storage by finding the closest ordered pair and counting up or down to the desired value.

Table 1 shows a sample of their tabulated results for programs, 1, 2 and 3 using three different time cut-offs for the experiment, namely 10, 20 and 30 seconds, i.e. for the purpose of this scaled-down laboratory study the criterion "time-feasible" was replaced by time cut-offs fixed at these levels. The first two columns of the Table make plain that a relatively small increment of advice to the store (raising the core-occupancy from 5,150 to 7,000 bits) yields a relatively large gain in the performance of the system, roughly tripling the range over which the prime-counter can be evaluated within the time-limit. The last two columns show the even more dramatic gains obtained by adding to the fixed advice a program-incrementable dynamic dictionary of rote-knowledge.

Rote-knowledge is of course the lowliest and least interesting of all forms of knowledge. Watterberg and Segre go on to speculate whether, retaining the "naive program *versus* advice" dichotomy, higher-level concepts could in principle be incorporated in the advice, even to the level of Hardy and Wright's (1938) "prime number theorem": $\lim_{x \to \infty} \frac{f(x)}{x/\ln x} = 1$, where $f(x)$ = number of primes less than x.

Formalising "knowledge"

This same dichotomy between naive program and advice has been made the basis of a package, AL1, for generating chess end-game strategies from Tables. Preliminary accounts are available elsewhere (e.g. Michie, 1976a). The present concern is that this style of conveying human knowledge to programs demands more precise and quantitative ways than are at present available for talking

about the knowledge-content of programs and adjoined advice. As we build and unbuild the advice, adding, modifying, or deleting rote-entries, descriptions, heuristics, theorems and the rest, we would like to be able to say of each separate item just what benefit it confers in terms of added knowledge and what is its cost in terms of added store-occupancy. A method of cost-benefit analysis will be sketched. But first it is desirable to have a clean-cut basis for definition and measurement of "knowledge." When this ground has been cleared, the philosophy can be forgotten.

Philosophers of the knowledge problem have always agreed that at least those facts which are explicitly and retrievably stored in memory are "known." What of facts which are stored implicitly, retrievable only by deduction? The earliest treatment of the problem seems to be the opening passage of Aristotle's *Analytica Posteriora*. He recognizes three levels:

universal knowledge of the *class* of facts implicit in the stored premisses (e.g. that all triangles have angles equal to two right angles);

virtual knowledge of every particular such fact which is deducible from the universal but has not yet either been deduced or acquired directly from sense-impression (e.g. that some particular triangle, which we have not yet encountered, has angles equal to two right angles);

unqualified knowledge of a particular fact which *has* been so acquired (e.g. that some particular triangle, which we have encountered, recognized, and thought about, has angles equal to two right angles).

In respect of his "virtual" category, Aristotle brings up some particular cases of difficulty, including the following. Consider a student's "knowledge" of the results of evaluating the predicate "having angles equal to two right angles" for an individual member of the class "triangle." The evaluation is conceived as proceeding in three stages:

(1) the instance is presented;
(2) it is recognized as belonging to the class "triangle;"
(3) the conclusion is drawn that the predicate is true of this particular instance.

Aristotle comments: "For example, the student knew beforehand that the angles of every triangle are equal to two right angles; but it was only at the actual moment at which he was being led on to recognize this as true in the instance before him that he came to know 'this figure inscribed in the semi-circle' to be a triangle... Before he was led on to recognition or before he actually drew a conclusion, we should perhaps say that in a manner he knew, in a manner not.

"If he did not in an unqualified sense of the term *know* the existence of this triangle, how could he know without qualification that its angles were equal to two right angles? No: clearly he *knows* not without qualification but only in the sense that he *knows* universally..." For consistency with context we must read the last sentence as though it continued, "... that this is true of the class

	Case A	Case B	Case C
Question asked:	Is 19 prime?	Is 199 prime?	Is 19,999,999 prime?
Stage of transition from virtual to unqualified knowledge en route to stage (5)	(3)	(4)	In practice never

TABLE 2. Three cases of the attempt to retrieve facts about the primeness of numbers. The numbers in parentheses refer to labelled stages in the text.

of triangles, and hence *knows* virtually that it is true of this triangle."

Aristotle's notion of a state of knowledge as capable of evolving dynamically under the sole influence of internal operations is a suitable point of departure for computational approaches, such as the present theory or I.J. Good's "dynamic probability" (this volume). But Aristotle's definitional system is a little too restrictive and we shall now test it to destruction by dropping increasingly heavy weights upon his "virtual" category. Then we re-build it so as to explicate knowledge in an unrestricted computational framework.

Aristotle remarked on the virtual category as being in some sense ambiguous and seemed unhappy with it. Any sense of dissatisfaction or unclearness (" . . . we should perhaps say that in a manner he knew, in a manner not") was, we propose, rooted in his unfamiliarity with computational ideas, in particular the notion that some facts may require impracticably lengthy calculations for their demonstration.

Turning from triangles to integers, consider again the predicate "prime." We shall suppose that the student evaluates it for a given integer n by testing for divisibility by unit increments of i where $1 < i^2 \leqslant n$. We shall also suppose that the student possesses rote knowledge of a few primes. Now consider the following stages:

(1) he noted that a candidate instance has been presented;
(2) he recognizes it as belonging to the class "integer;"
(3) he matches it against rote memory and if successful he goes to (5);
(4) he attempts to deduce (by evaluating "prime") that the predicate is true of this particular instance;
(5) he gives the answer.

For three superficially similar questions the student's responses might be as shown in Table 2.

Unless our student is a rare calculating prodigy we shall in case C eventually weary of waiting for his answer. So how can we ever credit him with "unqualified" knowledge that 19,999,999 is prime? Not until he has completed his calculation, which in practical terms he never does.

Aristotle's system of inference involved only computationally light syllogistic forms. If he had clearly envisaged the possibility that some straightforward questions might so draw out the calculative chains as to preclude an answer within the questioner's lifetime he would surely not have allowed the "primes" student the same knowledge status as the "triangles" student. Neither shall we. Instead we sub-divide the facts which a man "virtually" knows into (case B) those which are both virtually and practically known and (case C) those which are virtually but *not* practically known. In our system the former will be accepted as "known" but the latter are definitely *not known*. Such facts might subsequently achieve "known" status either through a modification of the knowledge-base which sufficiently shortens their derivation chains (e.g. addition of a key lemma), or by modification of the system of inference itself in the direction of improved efficiency so that some previously impracticable derivations become practicable, or by independent acquisition from an external source.

The "known" is thus equated to what can (practically) be retrieved, and Aristotle's "in a manner he knew, in a manner not" is replaced by "if able to answer with acceptable speed he knew, otherwise not."

Now we tie this down to specifics, and propose a workable calculus for measuring the knowledge-content of computer programs and associated bodies of advice.

Numerical measurement of knowledge

We interpret knowledge as the ability to answer questions, and we equate question-answering to the evaluation of finite functions. All questions in our system are expressible in the form "What is the value of f(x)?" All "answers" take the form "y", where y is an element of f's co-domain. We diverge from Aristotle in supposing that for some x's in X the given evaluation device (whose "knowledge" of f we are concerned to measure) may deliver the corresponding y-value in acceptable time but for other x's not. This time-bound, which is set by the questioner, is denoted by the symbol β.

β need not be of the cosmological magnitudes which we earlier used to dramatise the notion of "time-infeasibility": for example in the context of speed chess the user might set β to a few seconds.

It now becomes meaningful, having specified a machine, a program, and a particular value of β (expressed either in time-units, or in number of computational steps) to speak about some given f in the following fashion:

f is β-evaluable for x_1;
f is β-evaluable for x_2;
f is *not* β-evaluable for x_3;
f is β-evaluable for x_4;
f is *not* β-evaluable for x_5;
.
.
.
etc.

Consider now the partial function $f_K: X \rightarrow Y$, which is defined only for arguments in that sub-set of X, X_K, for which f is β-evaluable, i.e. in the above example $\{x_1, x_2, x_4, \ldots\}$. f_K corresponds to the *known part* of f. We can write it, in the terms of the above example, as $((x_1,y_1),(x_2,y_2),(x_4,y_4),\ldots)$. Taking the ordering of X as given, the string $y_1y_2y_4\ldots$ conveys the identical information. To determine the information-content of this string viewed as a classical information-theoretic message, we regard the constituent symbols as having been sampled from an alphabet consisting of the set Y with probabilities given by the relative frequencies with which the various symbols appear as right-hand elements of f's function table: $((x_1,y_1),(x_2,y_2),\ldots(x_N,y_N))$. [Remember that for a many-one function, size (Y)<size(X) and hence $y_i = y_j$ for some (i,j)'s. For example, f might be a Boolean function.] Earlier we wrote

$$I(f) = -N \sum_{Y} p(y)\log_2 p(y) \qquad (1)$$

for the information-content of f, where the expression p(y) is to be evaluated exactly as we have just described. But for the information-content of a designated fragment of f, such as its "known part" f_K, this expression is not suitable. The trouble is that $-\sum_{Y} p(y)\log_2 p(y)$ is an *average*, namely the average information-content per symbol. This average is then multiplied by N to obtain the entire message's information-content. If we want to deal in fragments we need a formula which associates with *each constituent symbol* its own proper information-content. The information-content of any sub-message whatsoever can then be found simply by summing directly over its constituent symbols.

The information-content of an individual symbol has been termed by Samson (1951) its "surprisal." For the message's r^{th} symbol it is $-\log_2 p(y_r)$. We accordingly re-write I(f) as a sum of N surprisals, thus:

$$I(f) = -\sum_{i=1}^{N} \log_2 p(y_i) \qquad (2)$$

It now follows naturally that the information-content associated with f's known part, f_K, should be

$$I(f_K) = -\sum_{i=1}^{N_K} \log_2 p(y_i) + N^* \qquad (3)$$

Where the successive values i = 1, 2, 3, ... index the right-hand elements of the 1st, 2nd, 3rd, ... members of f_K (not of f). $p(y_i)$ is reckoned as before from the frequency of the symbol y_i in f's function table (not f_K's). We thus arrive at a definition of the amount of knowledge about f as equal to *the information-content associated with f's known part*, i.e.

$$K(f) = I(f_K) \qquad (4)$$

This identification can be criticised on the grounds that the simple summa-

*N is the number of bits required to specify the binary condition "known/unknown" for each of the N members of f's domain.

tion of surprisals attaches equal weight to each, whereas the answers to some questions may be more useful to the questioner than the answers to others. From this standpoint a program for evaluating the chess function should surely not receive credit for its knowledge of the game-theoretic values of the "obvious" cases, which occupy most of the state space, on a scale equal to the knowledge which it displays when questioned on "interesting" positions of the kind which might arise in actual play. We meet this objection by refining (4) above, replacing it with

$$K(f) = -\sum_{i=1}^{N_K} u_i \log_2 p(y_i) \tag{5}$$

where the u_i's are utilities, normalised to have unit mean.

Finally, we relate benefit, in the form of useful knowledge as just defined, to the bit-cost of storing the program, L(f), and obtain the program's "computational advantage"·

$$C(f) = \frac{K(f)}{L(f)} \tag{6}$$

Note that for a time-minimal representation as a minimally encoded look-up table, C(f) is equal to unity, (since if $f_K = f$ then $K(f) = I(f) = L^*(f)$, where L^* is the bit-cost of the minimally (non-redundantly) represented table).... In what follows we drop the constant argument f and for convenience write K, L, . . . etc.

Knowledge-content of advice: a worked example

Our purpose in setting up this formalism is to be able to say, when a given body of advice p_B is added to the store and enabled to communicate with a naive solution program p_A, whether the advice has done some good, and how much good. To determine this we measure the cost-benefit parameters K and L both for the augmented program p_A plus p_B, and for p_A alone. The differences give us K_B, the amount of p's knowledge about f relative to p_A, and L_B, the bit-cost of p_B. K_B/L_B is then a cost-benefit ratio associated with the given body of advice. We refer to it as the system's "advisory advantage."

Actually it is a little more complicated than has just been suggested, because we must also take account of the cost-benefit parameters of an unavoidable "extra," namely the control program p_C needed to mediate communication between p_A and p_B. Strictly, the measurements of K and L for the unadvised program must be made with p_C loaded, and it is the measurements on p_A plus p_C which are to be subtracted to arrive at the final quantities for p_B. The relevant relations are

$$K = K_A + K_B + K_C \text{ (overall knowledge)} \tag{7}$$

$$L = L_A + L_B + L_C \text{ (overall store cost)} \tag{8}$$

$$C = \frac{K}{L} = \frac{K_A + K_B + K_C}{L_A + L_B + L_C} \text{ (overall computational advantage)} \tag{9}$$

	β	Program 1 A	Program 1 B	Program 1 C	Program 2 A+B+C	Program 3 (25 trials)	Program 3 (250 trials)
K	10 secs.	17×10^3	45.3×10^3	$-.3 \times 10^3$	62×10^3	$.95 \times 10^6$	6.0×10^6
K	20 secs.	33×10^3	85.5×10^3	$-.5 \times 10^3$	118×10^3	1.74×10^6	10.7×10^6
K	30 secs.	47×10^3	116.0×10^3	-1.0×10^3	162×10^3	2.38×10^6	14.6×10^6
L		5.15×10^3	$.25 \times 10^3$	1.6×10^3	7.0×10^3	12.1×10^3	26.5×10^3
K/L	10 secs.	3.3	181	–	8.8	78	226
K/L	20 secs.	6.4	330	–	16.9	144	402
K/L	30 secs.	9.2	464	–	23.1	196	549

TABLE 3. Knowledge, bit-costs and computational advantage measurements for Watterberg and Segre's programs 1, 2, and 3. Columns B and C were obtained by running program 2 with no advice, subtracting from column A+B+C to obtain column B and subtracting column A to obtain column C (see text).

K_C is always negative, and can be thought of as the "knowledge-overhead" associated with p_C. It is measured by running p_A with and without p_C and taking the difference between the amounts of knowledge measurable in the two cases.

Returning to the experiment of Watterberg and Segre we reproduce in Table 3 the values of these quantities as measured for the "prime counter" program with and without various increments of advice.

The tabulation brings out very clearly the feasibility and attractions of partitioning knowledge and costs parameters among different subdivisions of store, according to the "Incremental Advice" approach discussed earlier. In the last columns of the Table the impact can be noted of enabling the program to increment its own advice as part of an elementary rote-learning scheme. A finer partitioning of advisory knowledge and cost (not shown here) into components specifically associated with the rote-dictionary as distinct from the fixed body of advice reveals that the transition from 25 to 250 learning trials, although conferring almost a threefold further increase in overall computational advantage is approaching a point of diminishing returns: the computational advantage associated strictly with the rote-dictionary starts to fall. For details the reader is referred to the original paper.

As an aid to gaining an intuitive grasp, we should briefly consider the meaning of the K/L ratio when it characterises a body of advice, or an increment added to an existing body of advice. Clearly the ratio relates the amount of additional knowledge to the additional store-cost. But there is another way of looking at it which may possibly be illuminating. K_B/L_B is actually a ratio of compression. It tells us how much *less* store is consumed by adding the given knowledge to the system in the form of the given advice than would be consumed if the same knowledge were added in the form of a minimally (non-redundantly) coded lookup table for the partial function f_B. f_B of course consists of just those additional (x,y) pairs which become "known" when p_B is added to the system, these not having been "known" before .

Measuring a problem's "difficulty"

The earlier discussion of "semi-hard" problems did not extend to the quantitative measurement of degrees of hardness for such problems. The formalism which we have sketched now places us in a good position to do this. For a given f we consider the length of the shortest program possessing complete knowledge of the function: *not*, be it noted, the shortest program possessing complete information about the function; this would be equivalent to the function's Chaitin-Solomonoff-Kolmogorov "complexity" (*see* Chaitin, 1975), and since we have seen that for chess this quantity comes out to a mere 10^4 bits it plainly will not do as an index of hardness. Instead we take the function's *β-complexity*, i.e. the minimum bit-cost of a program possessing complete knowledge of f, I(f) in amount, and we call it f's "difficulty." If it comes to a very large number, corresponding to an infeasible store size, we say that f is "too difficult" for the

given β. Note that a "semi-hard" problem may be "too difficult" or it may be "not too difficult." Observations on the performance of chessmasters, and calculations as to the largest L_B that could be input to the human brain in a lifetime within the known limits to rates of information-transfer within the nervous system, encourage the hope that chess may turn out after all to be "not too difficult."

Knuth discusses the strategy when faced with a hard problem of accepting an evaluation mechanism with a bounded level of error. Extensions of the formalism presented here deal with certain forms of erroneous evaluation, and also with the notion that some degree of knowledge can be attached to a computation which is truncated by the β cut-off before completion and yet has succeeded during that time in shrinking the set of candidate answers. These and other details and elaborations will be found in the full account of the theory (Michie, 1976).

Note on "semi-hard" problems

It will reasonably be objected that the definition of "semi-hard" is needlessly disabled from corresponding with most people's idea of a machine intelligence problem, as a result of its critical dependence on concepts of "space-minimality" and "time-minimality." No one in the real world tries to make his program the shortest possible, nor would a sane man cling to time-minimality at the expense of a huge look-up table if by a small relaxation in running times he could obtain major savings of store. Problems may therefore exist which are easily soluble by conventional programming approaches, yet which can be made to look horrendous by rigid application of our definitions.

The remedy is to allow whoever wishes to use these definitions the freedom to blur their edges. Let him substitute "almost minimal" for "minimal" where he pleases, together with whatever tolerance in approximating the minimal (within a factor of 2, within a factor of 10, etc.) seems to him appropriate. Obviously the spirit of "semi-hard" is not met (nor does the need arise in such a case to assemble in store large bodies of advice) by a function whose trade-off curve is such as to permit reasonably fast evaluation by a near-minimal program, but for which an abrupt transition to time-infeasibility occurs as soon as the program is required to be actually minimal. We cannot ignore the possible prevalence and practical importance of such functions. Strictly, then, a definition which aims to be useful should filter them out.

Future work

The game of chess offers a domain which is finite, formally defined, at least "semi-hard," possibly "hard," and (most important) readily decomposable into sub-domains which can be isolated for separate study and measurement. Evaluation mechanisms exist in the brains of Grandmasters. Although not infallible, these can evaluate chess positions over most of the domain with an impressively low level of error. Studies by cognitive psychologists have shown these mech-

anisms to be "advice-driven" rather than "search-driven" and a massive literature has accumulated over centuries in which the masters have attempted to describe this advice. A challenge exists here to the machine intelligence specialist to translate and elaborate chess advice into machine representations which are precise and complete by the test of correct play against masters. A pure look-up (and correspondingly bulky) program for King, Rook and Pawn *versus* King and Rook written by Arlazaroff was recently validated in such a test, adjudicated by Grandmaster Awerbach (see *Firbush News 6*, (ed. J.E. Michie), Univ. of Edinburgh). Store-occupancy was of the order of a thousand megabytes. Systematic application to this domain of the framework which we have developed might be rewarding, proceeding backwards from the end of the game through the "foothills" as it were, i.e. via KRK, KPK, KQK, KQKR, KPKR, etc., these being specimen sub-domains into which KRPKR decomposes by loss or promotion. The thrust should be towards numerically characterising information-content and difficulty-bounds of individual sub-games, and measuring parameters relating to knowledge, cost and store-compression for each incremental component of advice. A number of examples and discussions of "foothill studies" have been brought together by M.R. Clarke in a recent book (*Advances in Computer Chess 1* (ed. M.R. Clarke), Edinburgh University Press).

REFERENCES

Chaitin, G.J. (1975) Randomness and mathematical proof. *Scientific American*, **232**, 47-52 (May, 1975).

Clarke, M.R. (ed.) *Advances in Computer Chess 1*, Edinburgh: Edinburgh University Press, 1977.

Fischer, M.J., reviews relevant network complexity results in *MAC Memorandum No. 65*, M.I.T., 1975.

Hardy, G.H. and Wright, E.M. (1938) *An Introduction to the Theory of Numbers*. Oxford University Press.

Knuth, D.E. (1976) Mathematics and computer science: coping with finiteness. *STAN-CS-76-541*. Stanford University.

Michie, D. (1976) Measuring the knowledge-content of programs. *Technical Report UIUCDCS-R-76-786*. University of Illinois, Computer Science Department.

Michie, D. (1976a) AL1: a package for generating strategies from tables. *SIGART Newsletter No. 59*, 12-14.

Samson, E.W. (1951) Fundamental natural concepts of information theory. *Report E5079*, Air Force Cambridge Research Station.

Schnorr, C.B. (1975) The network complexity of equivalence and other applications of network complexity. *Fachbereich Mathematik*, Universität Frankfurt.

Shannon, C.E. (1950) Programming a computer for playing chess. *Phil. Mag.* **41**, 256-275.

Watterberg, P. and Segre, C. (1976) Knowledge measurement. *CS397DM Group Report*, University of Illinois, Computer Science Department.