If intelligence is uncomputable, then…[*]

Peter Kugel
Computer Science Department, Boston College

Intelligent behaviour presumably consists in a departure from the completely disciplined behaviour involved in computation, but a rather slight one, which does not give rise to random behaviour, or to pointless repetitive loops.

Alan Turing [1]

Summary

I want to propose an abstract model of the human mind that is, to the human mind, a bit like what a Euclidean rectangle is to a farmer's field. I want to show why intelligence, in this model, requires a particular kind of uncomputability – the kind that Gold [2] has called computability *in the limit*. This observation has implications for Computer Science and Cognitive Science if my abstract model really captures the essence of intelligence.

0 How I got here and where I am going

This session brings together people who came to the conclusion that we need machines more powerful than computing machines from different starting points. I got here from the mind. It seemed, to me [3], that computing machines – good as they are – are not powerful enough to characterize *everything* the mind can do. Some of it, perhaps. But not all.

One thing they seem to do a bad job of characterizing is intelligence. I've argued that they can't handle Mozart very well [4] and, in this paper, I'd like to argue that they can't handle Einstein very well either. That's hardly surprising if you think about how Turing developed the idea of what we call the *Turing machine*, in terms of which the idea of a *computation* is usually defined. Turing was not trying to characterize the mental machinery of Mozart or Einstein. He was trying to characterize the mental machinery of people like Bob Cratchit, the clerk of Dickens's *A Christmas Carol*. As they might have put it in the 1930's, he was trying to characterize the basic capabilities of "computers".

Not the machines, of course. They hadn't been invented yet. In Turing's day, "computers" were people who carried out routine calculations to produce values for mathematical tables or, like Cratchit, for business records. Turing machines do a bang-up job of characterizing what such human "computers" (and today's electronic ones) can do. Most people agree that it is good enough to justify Turing's (and Church's) famous thesis which claims that the Turing machines – in spite of the very limited set of primitive

[*] Presented at a Special Session, *Beyond the Boundaries of Computability*, at the 2003 Spring Western Section Meeting of the American Mathematical Society, San Francisco, CA, May 3- 4, 2003

operations they can perform – can do anything a "computer" can do as long as it (or he or she) sticks to computing.

That's fine. But what about the capabilities of people who seem to do more than compute? How about the people who dream up the instructions that the human "computers" of Turing's day (and the electronic ones of ours) carry out? Can the Turing machine characterize what they do? I want to try to argue that they can't.

Let me sum up what I am going to say in the abstruse shorthand of mathematics, for those who understand that sort of thing, before I say it in what I hope is a more intuitive way. (If you don't understand this shorthand, you can skip the next paragraph.)

Computers – both human and electronic – can be thought of as evaluating the *totally computable* functions – functions that are in Σ_0 of the Kleene hierarchy [5]. Mathematicians, who prove theorems in formal systems, can be thought of as evaluating functions in Σ_1 of the same hierarchy – the *partially computable* functions. Turing machines can evaluate both kinds. Gold [2] suggested that the capabilities of people who derive general theories from specific examples might be better characterized by the ability to evaluate functions in Σ_2, a higher level of Kleene's hierarchy that includes functions less computable (or more uncomputable) than any in Σ_1. These functions can be said to be *computable in the limit*. I have argued [6,7] that intelligence requires the ability to evaluate functions that are computable in the limit, but not computable – functions that are, in other words, beyond the classical boundaries of computability. And I want to explain why I believe that machines limited to classical computing are, in a sense that I will make relatively precise, "stupid".

I want to frame my argument in terms of an abstract model of the human mind that is, to the real human mind, roughly what a Euclidean rectangle is to a farmer's field. As Euclidean rectangles ignore the wheat, the weeds and the worms of real fields, so my model of the human mind ignores most of the messy features of the real mind in an attempt to characterize its underlying structure. Within that model, something that looks an awful lot like intelligence is quite easily seen to require more than computing.

But it is also easily seen not to require any new machinery. All it requires is that we use the machinery of the computing differently. And that's convenient for those of us who would like hypercomputers[*] (or machines more powerful than computing machines) to work with because it eliminates the "building problem". That's the good news. The bad news is that it raises a "using problem". Unlike most of the other designs for hypercomputers proposed at this meeting, it's easy to *build* the kind of hypercomputer that I claim intelligence requires, because it's already been built. But it's not so easy to figure out how to *use* it effectively.

[*] The term "hypercomputers" was suggested by Copeland and Proudfoot [8].

1 My "spherical cow" model of the mind

There is an old joke about a dairy farmer who asks a mathematician for help in improving his farm's milk production. The mathematician studies the farm for a few weeks and reports back to the farmer. "Imagine that the cow is a perfect sphere," he begins.

Well, of course, the cow is not a perfect sphere and thinking of it that way is not a particularly productive approach to improving the yield of dairy farms. (Which is why this story is a joke rather than an introduction to scientific dairy farming.) But developing abstract models, even though they ignore many of the details that preoccupy the practical among us, can be productive if the models chosen are good ones. And they are good if they get at significant underlying structures and, perhaps more important, if we can use them to do something useful.

A computing machine (at a spherical cow level) looks more or less like this (Figure 1):

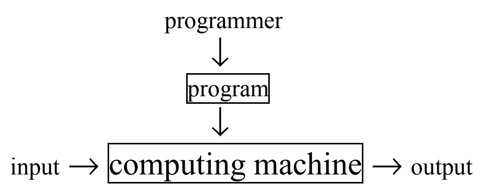

Figure 1
A computing machine

Now obviously a mind can't just be the computing machine part of this model. It needs a programmer to produce its program and, since we don't program each other, that programmer is, presumably, inside the mind. In other words, the situation looks more like this (Figure 2):

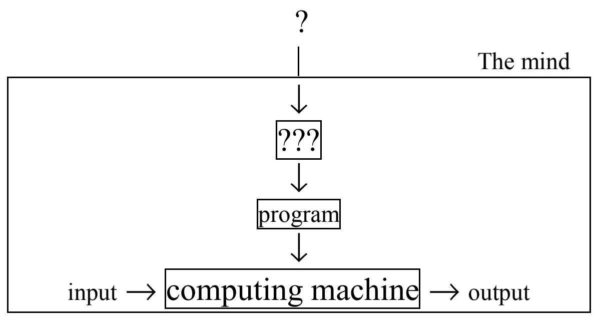

Figure 2
First cut at a model of the mind

But what does the $\boxed{???}$ component generate its programs from? I am going to assume – and I think that this is quite plausible – that it generates its programs from examples, much as a child develops the ability to understand an infinite set of utterances in its native language from the finite set of utterances that it hears[*]. Because I want to develop a very abstract model that ignores the messy details – I am going after what we might refer to, following Chomsky [10], the *competence* that underlies intelligence – let's assume that the computing machine in this model deals only with functions to and from the positive integers.

Let's assume that the inputs of the $\boxed{???}$ system of Figure 2 are the values of a function f – f(1), f(2), (f(3),… – from and to the positive integers, and that the output is a program, P_f, for computing the values of f. We might think of these inputs as being produced by a system in the mind's environment that implements the function.

To simplify the mathematics, let's assume that the values of that function are presented to this system in the form of a one-way infinite tape whose first square contains the value of f(1), whose second square contains the value of f(2), and so forth. (See Figure 3, below.) Because a machine can use such a tape to answer arbitrary questions of the form "What is the value of f(x)?" for a given function f, I propose to call it an *oracle* for f. (The name comes from Turing [11].) And, because we can think of the system that turns the values of the function into a program as compressing those infinitely many values into a finite program, I propose to call that program-producing system a *compressor*. The computing component of this idealized mind then expands the program generated by the compressor into the infinitely many values of the function. Because of this, I propose to call the computing component an *expander*.

The resulting system is what I like to call a *compressor/expander* (or *c/e machine*). Such a machine looks like this (Figure 3):

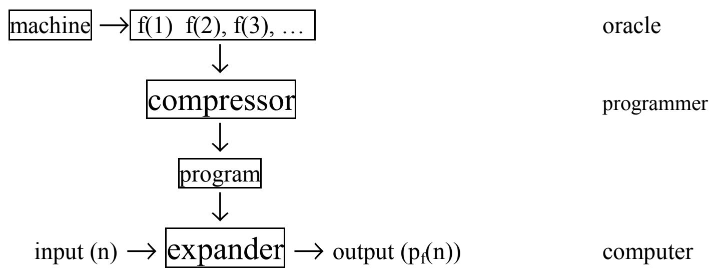

Figure 3
A compressor/expander or c/e machine

At first, it may seem that such c/e machines do not do anything useful. They compress only to expand. Seen as a whole, they take in the values of the function, f, and output them – not always correctly. But, in fact, such machines produce two things that that computer programmers call "side effects" that are rather handy. Compressor/ expanders can, as their name suggests, compress their inputs. Because their intermediate products (the programs they produce) are finite objects, they can, unlike the infinite data they characterize, fit inside the human skull. There will come a point when that program is more compact than a list of all the values seen "so far" is more compact than its representation as a list. Therefore, turning a list into a program can be a good way to remember it.

[*] The story is actually more complicated than this [9].

But a second side effect is even more important. If you ask such a system to produce its guess of the next value of f(n) before it has "seen" that value, its output can be used to predict its input. If the function it is compressing and then expanding represents the behavior of something important in its environment – an antelope it might like to eat, a lion that might like to eat it, or another c/e machine in which it might have a romantic interest – the expander's predictions of that something's future behavior could help the owner of the c/e machine eat, avoid being eaten and improve its love life.

The compressor components of c/e machines have other uses. The mind of a young child might use a compressor to go from the finite set of utterances in its native language that it hears, to programs that its expander might then use to produce new sentences in that language. The mind of a scientist might use a compressor to develop general theories from specific evidence. And a computer might use a compressor to generate its own programs from examples of what those programs are supposed to do.

2 Why compressors should do more than compute

There are good reasons why compressors should do more than compute. Look, for example, at what happens when they deal with what have come to be known as *black box identification problems*. In a black box identification problem a compressor is given a machine (or "black box") selected from some set of machines.

We say that a compressor can *solve a black box identification problem* for a given set of machines if it can correctly "identify" any machine in that set from its input tape alone, using a single algorithm. And by *identify*, I mean "find a program that generates the input tape".

With that in mind, let's look at what happens when we ask a compressor to identify (or compress) an arbitrary finite automaton drawn from the set of all finite automata that generate infinite sequences of 0's and 1's. We feed a tape representing the output of such an automaton into a compressor and ask it to come up with a program that will "predict" all the symbols of this output (See Figure 4).

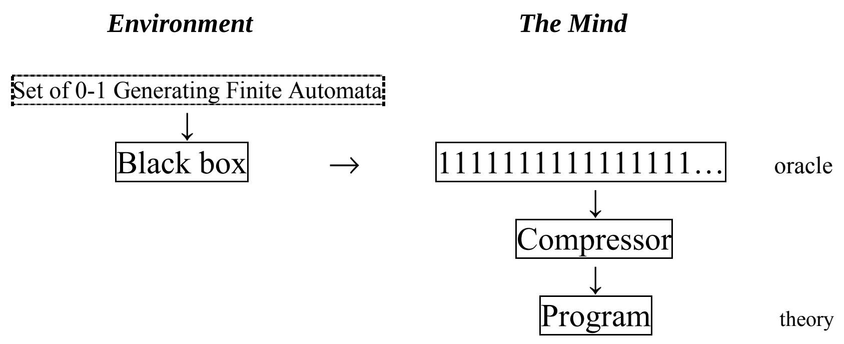

Figure 4
A compressor used to develop a theory of 0-1 generating finite automaton

Now consider what happens when we give such a compressor the tape that consists of all 1's. (Obviously, there is a finite automaton that generates such a tape.) If the compressor is a computing machine, it is allowed only one shot at generating a program for that tape. At some point it has to say that it has computed its (one and only) result. And, at that point it might as well stop – since it isn't allowed to "change its mind". Suppose that, in dealing with the tape of all ones, it does that after it has seen ten symbols. (It doesn't matter, for this argument, whether that number is ten or ten million, but ten is easier to visualize.)

The good news is that it got to the right program quite quickly. The bad news is that it will have to deal with any other tape that begins with ten 1's in the same way. (That's what it means for it to use a single algorithm. Same input, same output.) Which means it can't identify any other zero-one-generating finite automaton (or *Zogfa*) whose first ten outputs are 1. Unfortunately, there are infinitely many such Zogfas, none of which this algorithm can identify.

Let's call the set of all machines a compressor can identify its *scope*. The compressor that guesses "all ones" after seeing ten consecutive initial ones cannot identify any other machine whose initial sequence is ten 1's and I call such a set machines (or oracles) a *hole* in the scope. Clearly the scope of any computable compressor working with Zogfas must have such holes.

Holes are one drawback of computing compressors. Another is that, because they are allowed only one shot at coming up with a program, they are forced to be what we might call *stupid*. Suppose, for example, we feed the same machine a tape that starts with ten 1's and has only 0's from there on out. No matter how many 0's the compressor/expander sees (after the first ten 1's), it must insist that the next value is going to be 1. That's the inevitable consequence of using a computation for black box identification and, if that's not stupid, I don't know what is.

Computations may be OK for the expander of a c/e machine (although I am not sure that's true either [3]), but they make little sense for the compressor. Fortunately there is an alternative.

3. Computing in the Limit

When we use a machine to compute, it receives an input and then goes through a series of well-defined mechanical steps. When it produces its first output that output is taken to be its result and the machines stops (or turns to another job). In other words, when we use a machine to *compute*, we count its first *output* as its *result*.

Suppose that, instead, we count its *last* output as its result. If we decide to do that, we may have to let the machine keep running because we cannot be sure that its current output will be its last. As a result, it cannot, in general, satisfy the *announcement condition* – which requires that a machine announce when it has found its result. Although that makes such a procedure useless for many purposes, it does take it "beyond the limits of computation".

It becomes what is often called (following Gold [2]) a computation *in the limit*. Following Putnam [13], we might call a machine that has the machinery of a Turing machine but is allowed to compute in the limit (as a Turing machine is not) a *trial-and-error machine*.

It is not hard to show that a trial-and-error machine can do things that a machine limited to computing cannot. For example, it can solve the halting problem which Turing [14] proved, a machine limited to computing cannot.* Here's one way it might do it:

Trial-and-error solution to the halting problem: Given a program, P, and an input i, output NO (to indicate that P, running on i, or P(i), won't halt). Then run a simulation of P(i). (Turing [14] showed that such a simulation is always possible.) If the simulation halts, output YES to indicate that P(i) really does halt.

Clearly the last output that this procedure produces solves the halting problem, if you're willing to accept results arrived at "in the limit". Which proves that a trial-and-error machine is not a computing machine because it can do at least one thing a computing machine cannot.

But, of course, if you're seriously interested in solving the halting problem, perhaps because you want to detect cases in which a program won't output any result at all, this "solution" is silly. Although you can use its YES answers for this purpose, you can't use its NO answers because they do not satisfy the announcement condition. The machine cannot tell us, in general, when a NO is its final result. (If it could, then the halting problem could be solved by a computation.)

But limiting computations can be used for other purposes, such as dealing with black box identification problems in which the announcement condition cannot reasonably be met. (If you are trying to "identify" a machine from its infinitely many outputs, there is no way you can be sure that you have the right theory after seeing only finitely many outputs because your result goes beyond the information given.) The reward for giving up the announcement condition is avoiding stupidity. Unlike a computation, a limiting computation can change its result when it sees that its current one is wrong. If, for example, it sees ten 1' followed by a million 0's, it can "change its mind" and start predicting 0's at any time.

By using limiting computations, a compressor can also avoid holes in its scope. And it can do more. It can solve the black box identification problem for all Zogfa's. It can do this by using what Angluin and Smith [12] have called an *enumeration method*.** Such a method would be based on a computable enumeration, a_1, a_2, a_3,…, of programs for all possible Zogfas. Given such an enumeration, the enumeration method works like this:

Enumeration method: It starts with the first element of the enumeration (a_1) as its theory and predicts the first symbol a_1 generates. As it reads each symbol of the input tape, it checks to see if that symbol matches the predictions of its current theory (a_i). If it does, it sticks with its current theory and reads the next symbol. If it does not, it moves down the enumeration until it finds a theory (a_{i+j}) whose values are consistent with all the values it has seen so far.

It is not hard to see that every incorrect theory will eventually get discarded by such a procedure and that a correct one, once it is reached, will never get discarded. Although the problems of determining whether or not the current theory is correct, and of finding a replacement for it if it is not, are both totally computable, the process as a whole is not and it finds the right theory for every possible Zogfa in the limit. What's more, it

* Contrary to the conventional wisdom, Turing did not write about the halting problem. He wrote about a problem that is equivalent to it. I thank Jack Copeland for pointing that out to me.

** Suggested by Gold [2].

finds it in finite time, without using infinitesimals, and without using actually infinite space. Its users can't be sure *when* the right theory has been found, but they can be sure that it can handle any possible Zogfa.

The black box identification problem for Zogfas is not the only such problem this method can solve. It can solve the black box identification problem for any set of machines whose programs are all totally computable and can be totally computable. One problem it cannot solve is the problem of identifying an arbitrary totally computable function.

That is an important limitation on what can be done by a trial-and-error machine. An important limitation on what can be done by a computing machine is that it cannot solve any black box identification problem whose scope is *dense* in the sense that, if a finite sequence S is the initial part of any oracle in that scope, then there are at least two oracles that have S as their initial parts. Both of these limitations are easy to prove.

4. The intelligence of the scientist

Black box identification is an abstract (spherical cow) model of what scientists do when they go from evidence to theories. Popper [15] pointed out, long ago, that, in the developing general theories from specific evidence, scientists cannot meet the announcement condition without risking the informal equivalent of stupidity. They cannot, he argued, prove the correctness of their theories with the kind of finality with which mathematicians prove the correctness of their theorems. No matter how much favorable evidence they have seen, they cannot be sure – no matter how sound their reasoning – that their theories are correct. They may have seen a million black crows, but that does not mean that the million-and-first cannot be white.

Lacking effective proof procedures, scientists have to be satisfied with effective disproof procedures, holding on to a theory until it has been disproved. They prove their theories in the way the exception "proves" the rule – by testing it. As a result, they cannot satisfy the announcement condition in principle. There is no way they can effectively determine when their current theory is the last one they will ever need.

A compressor/expander, working on a black box identification problem, is a mathematical model of this kind of process and it strikes me as a useful one, not only of the scientific method, but also of the methods by which the people who "programmed" the human computers of Turing's day, and those who program the electronic computers of ours, come up with programs from examples of what the programs are supposed to do.

The Turing machine gives us a precise model of the things that "computers" and theorem-proving mathematicians do. I want to suggest that a trial-and-error machine gives us a mathematical model of the things that programmers, scientists and children do when they come up with general programs, theories and concepts from particular examples.

We already know quite a lot about this model from a theoretical point of view. For a rather old, but quite readable, survey of the state of knowledge almost twenty years ago, see Angluin and Smith [12]. For accounts of more recent results, see Osherson, Stob and Weinstein [16], Martin and Osherson [17], or the annual Proceedings of the Annual Workshops on Computational Learning Theory.

5. So what?

We have known about computing in the limit and trial-and-error machines for a long time – at least since Gold [2] and Putnam[13] introduced the basic idea in 1965. Gödel [18] used the idea to prove the completeness of the predicate calculus in his doctoral thesis in 1929. And Turing [19] wrote (albeit vaguely) about it in 1946 (in a passage that I will quote near the end of this paper).

In view of that, isn't it time for us to try to do something useful with it? I suspect that, unless we come up with some practical applications, computing in the limit will continue to remain of interest only to a few mathematicians who will grind out more and more theorems about it.

The world seems to take the usefulness of theories rather seriously. I suspect that the main reason we are talking about going beyond the limits of the Turing machines at this meeting, rather than about going beyond the limits of lambda definability or general recursiveness, is that Turing was able to put his model of computability to use while those who studied the others stuck largely to theorem proving.

There are many things we could use the idea of computing in the limit for. In Computer Science, for example, it might help us develop systems for doing what has come to be known as *programming by example.*

When a recent house guest asked me to tell him how to make a cup of coffee, I demurred. "I'd rather show you than tell you and then, the next time you want a cup of coffee, you can just do as I did." That's the idea behind programming by example. Instead of *telling* a computer how to do something, step by painful step, we *show* it how to do it and ask the computer to "do as we did".

I want to suggest that the c/e machine model could help us develop systems to do that in much the same way that Chomsky's competence models have helped us develop systems that deal with human and machine languages. They might help set the basic ground rules according to which such systems are built.

Systems that learn by being shown already exist, but they have limited scopes. (Most of them use computing compressors.) If we allow compressors to compute in the limit (as I believe we should) we can get broader scopes. But Gold [2] has shown that there cannot be a single c/e machine that identifies any possible computing machine from its values. Thus theory sets a limit on what we can do and tells us that we should not try for a universal black box identifier. Instead, we should try to develop a variety of special-purpose systems that work in specific and limited areas.

Another thing theory tells us is that, if we are going to try to deal with a domain that is *dense* (in the sense defined above), we should not limit ourselves to computing systems. In such cases, and such cases are the ones we typically want to deal with, we need to allow computing in the limit.

It is quite possible that attempts to do programming by example have failed so far because they have tried too hard to stay "within the computational box". Perhaps the reason they have failed is very much like the reason why Skinner's [20] attempt to develop a theory of human language failed, at least by Chomsky's [21] account. That attempt failed, largely because it tried too hard to stay "within the finite-automaton box".

Our model tells us that trying to stay within the computational box while trying to develop a system that learns from examples is – in most domains – a road to "stupidity".

There are quite a few different kinds of programs that adjust what they do by changing parameters on the basis of their "experience". Game-playing programs that adjust heuristics based on their "experience" are one example. Neural nets are another. Genetic algorithms are a third. The use of such programs to continually adjust what they do as more and more experience rolls in are trial-and-error methods of sorts.. But I believe (and I suspect we can prove) that methods based on these ideas cannot have sets of functions whose set of oracles are dense as their scopes. In particular, I believe that no simple parameter-adjusting method can identify an arbitrary Zogfa in the limit. (I challenge those who advocate such systems to show that my belief here is wrong.)

The enumeration method can, but it does not do it well. The method of enumeration I describe above is terribly inefficient. It does a bad job of figuring out what program to consider next. It is not hard to come up with a more efficient method based on the observation that any oracle for a Zogfa is *eventually periodic* – that it must be of the form S(T)* (a finite sequence of 0's and 1's, S, followed by a finite sequence, T, repeated forever). A method that looked for repeating periods could be a lot more efficient at coming up with the "next" theory than one based on a fixed enumeration.

That is not the only way that identification could be made more efficient. It is not hard to believe that allowing the c/e machine to choose which symbol of the oracle to "look at" next might speed up convergence in certain domains and it ought to be possible to find programming languages in which the structure of the program more closely resembles the structure of the oracles in its scope. Kleene's [22] language of regular events could serve as the basis of such a language for the Zogfa's. Post's [23] production systems might provide such a language for a more general class of oracles. Different languages for describing the set of all possible hypotheses might produce different methods for picking the next one to choose when the current one goes wrong. There is plenty of room for new ideas here.

Precise definitions of *intelligence*, based on scopes and speeds of convergence might help provide more robust foundations for the study of intelligence in machines and humans . And we might want to factor in the properties of the expander when we compare intelligences. We might want to say that A is *more intelligent* than B if the programs it comes up with are faster or more compact than those that B comes up with.

So far, we have been limiting the expanders of c/e machines to computing machines, but there is no reason why they might not also be trial-and-error machines and workers in Artificial Intelligence might want to develop such expanders. One thing they might do is to help us develop better understanding programs. Human understanding of specific utterances seems to "change its mind" much as trial-and-error machine do. If, for example, the comedian Henny Youngman says "Take my wife," most of us interpret that (if we haven't heard this joke before) as meaning "Take my wife, for example." When Youngman changes that by adding "Please", we have little trouble in changing our understanding of what Youngman meant. Understanding may very well be limiting computable.

Gardner [24] has suggested that the human mind may have different kinds of intelligence – with different scopes and different efficiencies. If it does, the human brain might use a collection of compressing subsystems – each with a different scope. Given a particular problem, it might set several of these subsystems to work, leaving it to some sort of supervisory system (which could be called *consciousness*, if you like) to decide which of the results that these systems came up with to use in a particular case. If good programs based on this idea could be developed, it might give Artificial Intelligence, not only more power, but also something that at least vaguely corresponds to consciousness to work with. (It has always surprised me that consciousness, which many believe to play an important role in human intelligence, seems to have almost no correlates in the programs developed by Artificial Intelligence.)

The trial-and-error machine model even suggests a way that we might eventually make it possible for a computer to have something like "free will". As long as we use a computer as a Turing (or computing) machine, its results are determined by its program and it is hard to see how such a program can be considered to have "free will". What it does was completely determined by its programmer.

Some people have argued that programs could have free will if their behavior were based on a random element. But, if what a program does is determined by the flip of a (figurative) coin, that behavior may be unpredictable, but it is not really free because it is not under the program's control.

Computing in the limit offers a third alternative. The result that a trial-and-error machine computes in the limit *is* determined, but it cannot *be* determined by a computation. A computing machine cannot predict the final result produced by a trial-and-error machine any more than it can predict behavior based on genuine randomness. But now the behavior is, rather clearly, under the control of the computer. Is it under the control of the computer's programmer? Not if the computer's "experience" also helps determine how the program goes. Such programs would be driven by neither nature nor nurture alone. They would be driven by both and that might be what free will requires.

Whether or not this collection of (rather vague) ideas has any bearing on computer or human free will is an open question. But they suggest an interesting[*] problem we might face if we have to deal with computers that have been programmed by example. Today, it is hard to rely on what computers can do because we cannot always understand their programs or predict whether those programs will work correctly. That problem will be exacerbated if we allow computers to work as c/e machines and develop their own programs because their behavior will then depend, not only on their programs, but also on their "experiences" which will always be different. We will not, so to speak, be able to step into the same program twice.

If we had a machine that had learned to perform open-heart surgery by trial and error, we would probably not understand the program it was using. Would we want to such machines to operate on us?

I don't know.

6. Conclusion

As I promised at the start of this paper, I have suggested an abstract model of the mind. I have shown why I believe that intelligence, in this model, must be

[*] "Interesting" in the sense in which that word is used in the Chinese curse: "May you live in interesting times."

uncomputable in a certain (trial-and-error) way. And I have suggested some applications of these ideas to Computer Science, Cognitive Science and Philosophy.

I believe that Turing may have anticipated at least some of these ideas. At the start of this paper, I quoted Turing to suggest that he seemed to realize that intelligence requires more than computing. I also believe that he had at least a vague idea of what kind of "more than computing" intelligence might require. Thus, in what was probably his earliest recorded discussion of machine intelligence [19], he wrote that:

> There are indications … that it is possible to make the (Automatic Computing Engine) display intelligence at the risk of its making occasional serious mistakes. By following up this aspect, the machine could probably be made to play very good chess.

Turing was, I believe, right in his suggestion that intelligence requires the ability to make "serious mistakes". That is, after all, what a trial-and-error machine does. But he was wrong to believe that it would take a trial-and-error machine "to play very good chess". Today, we know that machines can play very good chess, even if we limit them to computations alone. They can be programmed to do it. But most people would probably agree that that fact only suggests that playing good chess does not require intelligence.

My guess is that it's *learning* to play good chess that does. But not *only* chess. Intelligence is broader than that. A system that only learns chess has a rather narrow scope. And, when we're talking about intelligence, scope matters.

References

[1] Turing, A.M. (1950) 'Computing machinery and intelligence', *Mind 59 (N.S. 236), 433-460.*

[2] Gold, E.M. (1965) 'Limiting recursion', *Journal of Symbolic Logic*, 30, 28-48.

[3] Kugel, P. (1977) 'Induction pure and simple', *Information and Control,* 35, 276-336.

[4] Kugel, P. (1990) 'Myhill's thesis: There's more than computing in musical thinking', *Computer Music Journa*l, 14, 1, 12-25.

[5] Kleene, S.C. (1943) 'Recursive predicates and quantifiers', *Transactions of the American Mathematical Society*, 53, 41-73.

[6] Kugel, P. (1986) 'Thinking may be more than computing', *Cognition,* 22, 137-198.

[7] Kugel, P. (2002) 'Intelligence requires more than computing...and Turing said so', *Minds and Machines*, 12, 4, 563-579.

[8] Copeland B.J., Proudfoot D. (1999) 'Alan Turing's forgotten ideas in computer science', *Scientific American* 280, 76-81.

[9] Bloom, P. (2000), *How Children Learn the Meanings of Words*, Cambridge, MA: MIT Press.

[10] Chomsky, N. (1965), *Aspects of the Theory of Syntax*, Cambridge, MA: MIT Press.

[11] Turing, A.M. (1939) 'Systems of logic based on ordinals', *Proceedings of the London Mathematical Society*, *Series 2,* 45, 161-228.

[12] Angluin, D and C. H. Smith (1983), 'Inductive Inference: Theory and Methods', *Computing Surveys,* 3, 237-269.

[13] Putnam, H. (1965) 'Trial and error predicates and the solution of a problem of Mostowski', *Journal of Symbolic Logic*, 20, 49-57.

[14] Turing, A.M. (1936) 'On computable numbers, with an application to the Entscheidungsproblem', *Proceedings of the London Mathematical Society, Series 2,* 42, 232-265.

[15] Popper, K. (1959) *The Logic of Scientific Discovery*. (translation of *Logik der Forschung*), London: Hutchinson.

[16] Osherson, D, M. Stob and S. Weinstein (1986), *Systems That Learn*, Cambridge, MA: MIT Press.

[17] Martin, E. and D. Osherson (1998), *Elements of Scientific Inquiry*, Cambridge, MA: MIT Press.

[18] Gödel, K. (1930) 'Die Vollständikeit der Axiome des logischen Funktionenkalküls, *Monatshefte für Mathematik und Physik* 37, 349-360.

[19] Turing, A.M. (1946) 'Proposals for the development in the Mathematics Division of an Automatic Computing Engine (ACE)' in Carpenter, B.E. and R.N. Doran (Editors) (1986) *A.M. Turing's ACE Report of 1946 and Other Papers*, Cambridge, MA: MIT Press.

[20] Skinner, B.F. (1957) *Verbal Behavior*, NY: Appleton-Century-Crofts.

[21] Chomsky N. (1959), 'A review of B. F. Skinner's *Verbal Behavior*', *Language*, 35, 26-58.

[22] Kleene, S. (1956), 'Representation of events in nerve nets and finite automata.
In Shannon, C. and McCarthy, J. (editors) *Automata Studies*, 3-42. Princeton, NJ: Princeton University Press

[23] Post, E. L. (1936), 'Finite combinatory processes -- formulation I', *Journal of Symbolic Logic* 1, 103-105.

[24] Gardner, H. (1983). *Frames of Mind: The theory of multiple intelligences.* New York: BasicBooks.