Uncomputers

Peter Kugel, Boston College
(kugel@bc.edu)

Summary

Like hypercomputers, uncomputers can evaluate uncomputable functions. But, unlike hypercomputers, they need not *compute* them. They can, to coin a phrase, "*uncompute*" them. Uncomputing has a disadvantage if you want a machine to solve the halting problem, but it has a compensating advantage – indeed, I shall argue, it's required – if you want a machine to behave intelligently.

0. Introduction

Like most of the people at this workshop, I believe that machines can evaluate uncomputable functions. But, unlike most of the rest of you, I don't believe that we will need new kinds of machinery (hypercomputers) to do that. I believe that the computers we already have will suffice. All we have to do is let them (as Turing,1950, suggested) "(depart) from the completely disciplined behaviour involved in computation", or (as I shall put it) "uncompute".

Because the name for computers in English[1] suggests that their machinery will be limited to computing, I will refer to computers that are allowed to do more than compute as "uncomputers"[2]. In this paper, I want to:

(1) Explain how one kind of uncomputer (a "trial-and-error machine") might work.
(2) Say something about why such machines might be worth looking into.
(3) Suggest some problems we might work on to help us use them effectively.

1. How to "make" an uncomputer

To see how the "disciplined behaviour" of computation might be relaxed to allow ordinary computers to evaluate (but not compute) uncomputable functions, consider how it might be done to "solve" a problem that I will call (incorrectly) "*Turing's halting problem"[3]:

**Turing's halting problem*: Find a single program, Halt, that will *determine*[4], for any program (Prog) with any input (Inp), whether Prog(Inp) (Prog running on Inp), will or will not halt.

Theorem 0 (Turing, 1936): No such program is possible if, by "determine", you mean "*compute with a regular computer*".

So, if you believe (as many of us here do) that a machine can solve *Turing's halting problem, you have two choices. You can try to do it by using machinery (*hardwar*e) that is more powerful than what is available in a regular computer, or you can stick with the

[1] The French call them "ordinateurs", presumably because they find the name "calculateurs" ("computers") too restrictive. (See http://fr.wikipedia.org/wiki/Ordinateur)
[2] I am not alone in my interest in what I am calling "uncomputers". It is shared by Mark Burgin (Burgin and Klinger, 2004) of UCLA, Dina Goldin (Wegner and Goldin, 2003) of the University of Connecticut, Kevin Kelly (2004) of Carnegie-Mellon, Peter Wegner (Wegner and Goldin, 2003) of Brown and almost certainly others. Our work shares a common ancestry in the pioneering work of Mark Gold (1965) and Hilary Putnam (1965). But like most "families", we squabble about some things.
[3] I follow the convention of the linguists by prefacing my improper usage by an asterisk.
[4] The proper statement of this problem substitutes the word "compute" here.

machinery of regular computers and allow that machinery to use methods (*software*) that can do things that computations cannot. Most of you probably prefer the hardware approach. I want to try to convince you that, for at least some purposes, the software approach is better[5].

Theorem 1: If all you mean by "determine" (in the statement of *Turing's halting problem) is "determine mechanically, using only finite time and space to get results", then the problem is solvable.

Proof: Consider the procedure (Halt) that goes like this. Given the inputs Prog and Int, Halt begins by outputting NO to indicate that Prog, running on Int, won't halt. Then it simulates the process of running Prog on Int, step by step, and, if the simulation halts, it outputs YES.

It is not hard to convince yourself that the last output this process generates is correct and that it is generated in finite time, using finite space. Following Putnam (1965), I call such procedures *trial-and-error procedures*. They differ from computations in that if a computation outputs a NO, it means it. When our trial-and-error procedure outputs a NO, it may not mean it because it might subsequently "change its mind". One way to put that is:

Definitions: When a computer performs a *computation,* we count its *first* output as its result. When it performs a *trial-and-error procedure* we count its *last* output as its result (but we may not know when an output is its last)[6].

A trial-and-error solution to *Turing's halting problem cannot be reduced to a computation (by Theorem 0), but it is a lousy solution because it does not meet the *eureka condition* in the sense that it does not always announce when it has produced a final result. Thus, when a computation tells you that 5+7=12, you know that 12 is its result. When a trial-and-error procedure tells you that a Prog(Inp) won't halt, you won't necessarily know when NO is its final result. Their failure to meet the eureka condition makes uncomputations worthless for many uses. But not for all.

2. How to use an uncomputer

There are problems for which uncomputations are preferable to either ordinary computations or hypercomputations precisely because they need not satisfy the eureka condition. Consider, for example, the problem of developing a theory from evidence. To repeat an example that I have probably used too often already, consider the problem of developing a theory that predicts the color of swans. No matter how many white ones you have seen, you cannot conclude (with finality), from the evidence at hand at any time, that all swans are white because the next one you see might always be black.

Generating theories from evidence is an example of the more general problem of *inverting computable functions*. Inverting a computable function is different from evaluating it. When we evaluate a computable function we go from a program for that function to its values. When we invert a computable function we go from its values to a program for computing it. So, for example, to evaluate the function Double, we go from 2 to 4, from 6 to 12, and so forth. When we invert a function, we go from f(2)=4, f(6)=12,

[5] It's not a question of either/or. It is possible to endorse both approaches because hypercomputers and uncomputers can serve different purposes, even though they both evaluate uncomputable functions. That is because hypercomputers *compute* them and uncomputers *uncompute* them.

[6] Another way to put it is to say that both computations and uncomputations require only finite time to produce their results, but computations are limited to computable amounts of time, whereas uncomputations are allowed to use uncomputable amounts.

and so forth, to a program that computes Double. It is easy to see that this process cannot satisfy the eureka condition. Let me try to be a bit more precise.

Let the *oracle* (Turing, 1939) of the function, f, be a one-way infinite tape whose nth square contains the value of f(n) for all n. In terms of such (infinite) oracles, we can think of the evaluation of a computable function as a process that goes from a program for computing a function to its oracle. And we can think of the inversion of a computable function as a process that goes the other way – from the oracle of a function to a program for computing that function.

The evaluation of a computable function is computable. Its inversion generally is not – not even by a hypercomputer. Here's why. If your procedure for inverting a computable function is to come up with a program, it must do so at a moment, t. But, at t, it can only have seen finitely many values of the oracle and the values it has seen at time t will not fully determine the function. So it will always be possible that it has come up with the wrong program at t because the program specifies values not yet seen. As psychologists put it, it "goes beyond the information given."

The inversion of computable functions suggests good models for what scientists do when they generate theories from evidence (Kelly, 2004), what programmers do when they generate programs from incomplete specifications (Kugel, 2005) and what children do when they learn general ideas from specific examples (Gold, 1965).

In my talk, I shall argue that intelligence is not only the ability to use good algorithms well (as is widely believed). It is also the ability to acquire them. (Kugel, 2002) Because acquisition is not computable (not even by hypercomputers) it requires uncomputers. So we will have to use uncomputers if we want to develop intelligent machines

For all these reasons, and more, uncomputations[7] deserve our attention.[8]

3. How to develop uncomputers

There are some problems we should probably try to deal with if we hope to make effective use of uncomputers. Some are consequences of a theorem about the domains of trial-and-error methods. The *domain* of a trial-and-error method, M, for generating algorithms from values, is the set of all functions whose algorithms M can correctly generate from their oracles.

Theorem 2: There is no trial-and-error method for generating algorithms from examples that is *universal* in the sense that its domain is the set of all computable functions. (Gold,1965)

This suggests the following problem:

*Problem 1***:** Develop methods whose domains are interesting sets of functions, such as the set of all functions that can be computed by finite automata (Kugel, 2004), the set of all context-free grammars, or the set of all winning chess games.

But, such methods can be notoriously inefficient so there is a problem that probably needs to be worked on before such methods can be used for practical purposes:

*Problem 2***:** Develop *efficient* procedures for inverting "interesting" sets of functions.

We might start with limited "toy" domains such as the problem of extrapolating numerical sequences or the problem of learning strategies for playing simple games by playing them. We might find it necessary to help such procedures by providing them with

[7] For a discussion of uncomputations other than trial-and-error procedures, see Kugel (1986).

[8] I find it amusing to think that we are here, trying to go beyond the so-called "Turing limit", when Turing machines already evaluate uncomputable functions. That they do so is easy to see. Since they evaluate all the partially computable predicates in Σ_1 of the arithmetic hierarchy, there must be some predicates that they evaluate without computing all their values.

more information than can be gleaned from an oracle alone. For example, humans often need the help of "how-to" demonstrations, hints, explanations or advice. This suggests another problem:

*Problem 3***:** Develop ways to "help" procedures for inverting computable functions.

Many interesting domains involve functions whose vales are not usually fed into computers, which suggests:

Problem 4: Develop methods for inputting examples from the "real" world.

There are, of course, lots of other problems we might work on, but let me end my list with a more general one:

Problem 5: Develop a mathematical theory of "computing in the limit", which is what trial-and-error procedures do, in the hope that it might play a role in the information sciences similar to that played by calculus in the physical sciences.

4. Conclusion

There are certain problems (such as the real halting problem) whose solutions cannot be computed by computers because their hardware is not powerful enough. Such problems might by solved computationally by the more powerful *hardware* of hypercomputers, if such hardware can be built. There are other problems (such as the problem of inverting computable functions) that cannot be solved by computations or hypercomputations because they have to satisfy the eureka condition by announcing when they are finished. For such problems, traditional computing machinery, using the less demanding *software* of trial-and-error procedures (uncomputations) may be more suitable.

Most of the people in this workshop are interested in hardware approaches to *computing* the uncomputable. I am more interested in software approaches to *uncomputing* the uncomputable and, in this summary, I have tried to explain why. Let me add a final reason for preferring this approach. We don't know whether we can actually build hypercomputers or not. But we do know that we can build uncomputers. They already exist on your desk and mine. All that we have to do is figure out how to use them effectively.

I am trying to do that. As a man I obviously admire wrote in a slightly different, but related, context (Turing, 1950): "We can only see a short distance ahead, but we can see plenty there that needs to be done." I invite you to join me in trying to do it.

References

Burgin, M. and A. Klinger (2004), "Experience, Generations, and Limits in Machine Learning", *Theoretical Computer Science*, 317.

Gold, E.M. (1965) "Limiting Recursion", *J. Symbolic Logic* 30.

Kelly, K. (2004) "Uncomputability: the Problem of Induction Internalized", *Theoretical Computer Science*, 317.

Kugel, P. (1986) "Thinking May be More than Computing", *Cognition*, 22.

Kugel, P. (2002) "Computers Can't Be Intelligent (and Turing Said So)", *Minds and Machines*, 12

Kugel, P. (2004) "Toward a Theory of Intelligence", *Theoretical Computer Science*, 317.

Kugel, P. (2005) "It's Time to Think Outside the Computational Box", *Communications of the ACM*, 48.

Putnam, H. (1965) "Trial and Error Predicates and the Solution of a Problem of Mostowski", *J. Symbolic Logic* 30.

Turing, A. (1936) "On Computable Numbers, With an Application to the Entscheidungs-problem", *Proceedings of the London Math. Soc., Series 2,* 42.

Turing, A. (1939) "Systems of Logic Based on Ordinals", *Proceedings of the London Math. Soc., Series 2*, 45.

Turing, A. (1950) "Computing Machinery and Intelligence", *Mind* 59, 236.

Wegner, P. and D. Goldin (2003), "Computations Beyond Turing Machines", *Communications of the ACM*, 36.