By PETER KUGEL

IT'S TIME TO THINK OUTSIDE THE COMPUTATIONAL BOX

Systems that don't announce when they've reached their final results can 'compute' the uncomputable, possibly allowing them to understand their users and generate their own programs from examples.

Many jokes begin with: A man walks into a bar and says "My wife doesn't understand me." Not many begin with: A man walks into a bar and says "My computer doesn't understand me." It's easy to see why. We don't expect our computers to understand us. We're supposed to understand *them*. That's why there are user manuals and programming languages. Does it have to be this way?

Computers would be a lot easier to deal with if they could somehow try to understand us. Notice, for example, how much easier it is to "program" children to recognize dogs than to program computers to recognize dogs. The problem is that computers need programmers to help

ILLUSTRATION BY JEAN-FRANÇOIS MARTIN

them understand us, and children do not (see Figure 1). If we want to program a child to identify dogs and distinguish them from, say, cats and trees, all we have to do is point to a few examples. Children try to understand what we are doing and develop a program (or something like it) that allows them to do likewise.

In contrast, if we want to program a computer to do it, a systems analyst must first figure out exactly what makes something a dog. A programmer then has to write a program that tells the computer—in one of its finicky languages—how to recognize them. The program must be checked out to make sure it works properly, and, by the time it is delivered, the child will have learned to use not only "dog" but thousands of other words.

Wouldn't it be nice if we could program computers the way we program children—by giving them examples of what we want them to do and letting them develop their own programs to do it?

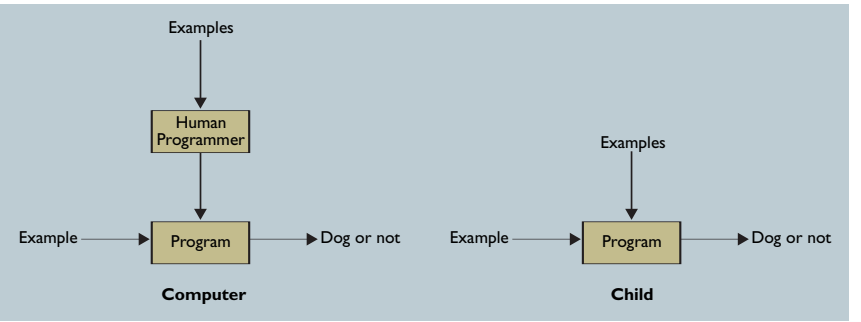

Figure 1. Computers need programmers; children do not.

This idea is not new. It's been called "programming by example" and is such a good idea that you have to wonder why more people aren't working on it. Perhaps the reason is that, at first glance, it seems impossible. But the fact that children do it shows it must be doable. Perhaps it seems impossible for much the same reason it seemed impossible (to the ancient Greeks) for the square root of two to be a number. It's not a number if you assume the only numbers are the rational numbers. But if you add the irrational numbers to your conceptual toolbox, the square root of two becomes a number. And, by expanding your idea of a number, you gain some powerful new tools.

Similarly, we may not be able to get computers to do a decent job of developing programs from examples if we limit ourselves to computations. We may have to allow them to use algorithms from outside the computational toolbox.

To see why, suppose you are trying to produce a program to classify positive integers (1, 2, 3...) from examples. Ask yourself how you might want such a system to learn to recognize gleeks (a nonsense word I made up because you're not supposed to know what gleeks are). In other words, suppose you wanted to develop a system that would go from examples of numbers that are gleeks and non-gleeks to a program that recognized gleeks it had not seen before.

You tell the system that 2, 4, and 6 are gleeks and that 1, 3 and 5 are not. What should it do next? Assuming there are no errors in the inputs, the system should, at this point, probably output a program `IS-EVEN` that classifies even integers as gleeks and odd integers as non-gleeks. However, even though the evidence seems pretty conclusive, `IS-EVEN` could still be the wrong program. Gleeks might, for example, be all the even numbers smaller than 8 or all the numbers whose names in English don't end in "e" or any of the infinitely many other possibilities consistent with the information given.

And—this is the point—if our algorithm for generating programs is a computation, it is not allowed to change its mind once it decides that `IS-EVEN` is the right program. It has to stick with `IS-EVEN` because computations are allowed only one conclusion per input. After all, when you ask a computer to compute your car payment, you expect it to output its result and stop. You don't want it to tell you that you owe $1,234 and then, a week later, tell you it has changed its mind, and you really owe $5,678. Computations are not, by definition, allowed to do that.

So you might want your system to wait longer before it outputs its final answer. But that won't do either, because no matter how long it waits, no matter how many examples it sees, there are always infinitely many possible different programs consistent with these examples, any one of which could be correct.

If we insist on limiting our program-generating systems to computations or to algorithms that come to a single conclusion for any given input and announce when they have done so, our systems will suffer from at least two significant limitations:

For each program they generate correctly from examples, there will be an infinite set of programs consistent with the same examples they cannot generate. We might call the set of all such programs, consistent with the information given but not derivable by our system, a "black hole" in its scope—an infinite set of computable functions for which it cannot derive a correct

program because it derives an incorrect one. For example, if such a system produces IS-EVEN after seeing the evidence for 1, 2, 3, 4, 5, and 6, then it is the only program it is allowed to come up with for that evidence. But there are infinitely many other programs consistent with the same evidence it cannot come up with, because it will generate IS-EVEN (as its only program) for them.

Any system that uses the programs output by our systems to predict future numbers as gleeks is "pigheaded" in the sense it will continue to stick to a program in the face of overwhelming evidence that it is incorrect. For example, if such a system generates IS-EVEN after seeing the first six examples (and gleek turns out to refer to "the even integers less than 7 and odd integers from 7 on up," a system using the system's output to predict gleekishness is committed to sticking with IS-EVEN (or being pigheaded), even though all its predictions for integers above 6 are incorrect.

These defects result from the fact that a computation is allowed only one shot at a result, and that it must "announce" when it has obtained it. Suppose, however, these restrictions were relaxed. Suppose we allow a computer to change its mind and do not require that it announce when it has produced its final result. If we do, the resulting system can do more than compute. It can use algorithms—referred to by Burgin in [1] as "super-recursive"—that can do the uncomputable.

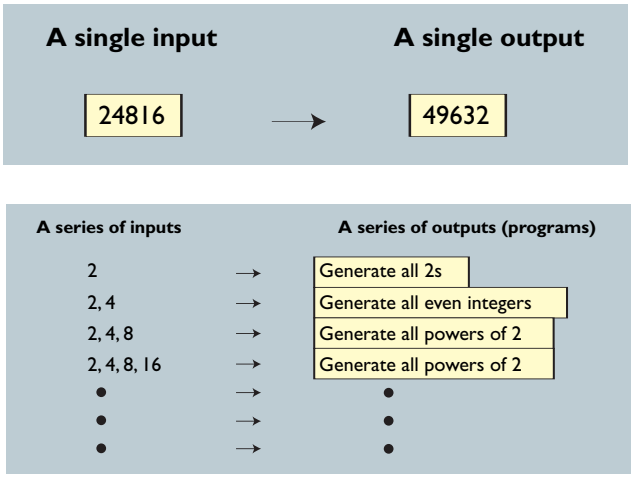

Figure 2. A regular computation (a) vs. a limiting computation (b); a regular computation has one input and one output, and a limiting computation has many inputs and many outputs.

Computing in the Limit

One class of super-recursive algorithms compute in the limit, or allow a finite number of incorrect guesses (see Figure 2) [2, 8]. When we use a computer to compute, we take its first output to be its result. But when we use it to compute in the limit, we take its last output as its result without requiring that it announce when an output is its last.

It is not difficult to show that, when they are allowed to use such algorithms, computers can solve problems they cannot solve through classical computations. For example, consider the halting problem, or the problem of developing a single procedure (HALTS) that tells us whether a given program (PROG) running on a given input (INP) will or will not halt. Alan Turing proved this problem cannot be solved through a computation [10]. It can, however, be solved by the following limiting computation:

Limiting-computing algorithm (HALTS) for solving the halting problem:

- *Output NO (to indicate that PROG running on INP will not stop).*
- *Simulate the running of PROG on INP, step by step, and, if the simulation halts, output YES.*

It is not difficult to see that the last output produced by this algorithm is always correct, proving that limiting computations can do things that classical computations cannot.

At this point, people trying to develop systems to generate programs from examples must choose. They can either disallow limiting computations on the grounds that they are not "genuine" computations (and computers must be limited to computations) or they can allow them on the grounds that computers are capable of using them. The dilemma is not unlike the dilemma that faced mathematicians in early Greece when they discovered that the square root of two could not be expressed as a rational number. The Greeks could have said that the square root of two is not a number (period) or they could have expanded their idea of number to include the irrational numbers and count the square root of two as a number.

Fortunately for us, they chose the latter, expanding the box of tools available to mathematics so it was ready for bigger things, such as calculus.

Similarly, allowing computers to "compute in the limit" lets them do new things, though at a cost. When you allow limiting computations, your results lose the kind of finality you get from the results of computations. In our "solution" to the halting problem, the YES results have the finality of computed results. However, the NO results may not. There is no moment (in general) when a computer generating a NO output can tell you, with finality, that PROG running on INP will not halt. This means that NO answers will be useless for program debugging. But it doesn't mean that limiting computations are useless.

If our algorithm for generating programs is a computation, IT IS NOT ALLOWED TO CHANGE ITS MIND once it decides a particular program is the right program.

They are useful for other things, most notably programming by example. Here is an algorithm that might be used to generate programs from examples:

Limiting-computing algorithm for programming by example [2]:

Given a list of totally computable programs, P_1, P_2, P_3,...

• Start by "guessing" the first program on the list, P_1—which is to say, outputting it.

• Each time you get another example, check to see if the latest program output, P_n, is consistent with that example. If it is, do nothing and continue reading examples. If it is not, go down the list of programs and output the first program that is consistent with all the examples seen so far.

It is not difficult to see that, if at least one correct program is on the list, and all the programs on the list compute a result for every input, the first correct program on the list will eventually be output (because all the programs earlier in the list will have been disproved by at least one counterexample), and that, once it is output, it will not be retracted (because further evidence will not disprove it). The good news is that this (super-recursive) algorithm can avoid the main problems of computational (recursive) algorithms, including the kinds of black holes a computational algorithm produces, as well as pigheadedness, precisely because it can "change its mind" [4]. Moreover, like a classical computation, it produces its final result in finite time.

The bad news is that we cannot tell when it has produced a final result, so our trust in its results is always tentative. For example, after seeing some evidence, it may infer that the gleeks are the even numbers. But, no matter how much evidence it sees in favor of this claim, it can never tell with certainty that all even numbers are gleeks.

Drawbacks

Many users might find systems that produce such wishy-washy results unappealing. But if we want to generate programs from examples, we may have to live with them. Because such algorithms always go beyond the information given, we cannot be certain (at any moment) that they have done it correctly.

That problem of determining whether the current theory is the ultimately correct one is familiar to scientists in all fields. If, say, ornithologists have seen a million swans, all of which are white, they cannot be certain that all swans are white. The million-and-first might very well be black. Popper's account of the scientific method in [7] is based on this observation, suggesting that, in contrast to mathematicians, scientists cannot produce conclusive proofs that their current theory is true. All they can do is produce conclusive disproofs that their current theory is false. As a result, they accept theories as true until they have been disproved—much as our super-recursive algorithm for programming by example accepts programs as correct until the evidence shows they are not.

Even if we are willing to accept such always-tentative results, there are at least two problems with such methods for developing programs from examples that have to be dealt with if we want to put them to practical use. They are incomplete in the sense that no such algorithm can learn all possible programs [2, 4, 6], and they are inefficient.

A mildly tricky mathematical argument [2] is required to show that these methods must be incomplete, but the basic idea is that, if an algorithm's list of programs (P_1, P_2, P_3,...) contained all possible programs, the algorithm would also have to contain some programs that do not compute a result for every input. When the algorithm considers one of these programs, with an argument for which it produces no result, that algorithm/evidence pair will not be disproved, and the algorithm will not move on to the next program on the list.

It's not difficult to see that an algorithm that blindly looks through such a list can be horrendously inefficient. To find a program that displays the message "Hello, World!" by looking through a list of all possible programs could take as long as waiting for monkeys, typing randomly, to produce *Hamlet*.

Applications

We will have to deal with these problems before we are able to use limiting-computable algorithms to generate programs from examples [4] or to develop systems that understand their users' behavior. One way to handle incompleteness is to accept it and make do with systems that work only in limited application areas. This is almost certainly a good way to start. We might be able to improve comprehensiveness by allowing systems to use super-recursive algorithms that are even more powerful than limiting-computing algorithms [1, 6], but such systems would involve additional drawbacks.

Overcoming inefficiency is probably more important than dealing with incompleteness. The rate at which an algorithm gets to the right program might be speeded up by either using knowledge about particular application areas or allowing the algorithms to rearrange their lists as more information about the current problem comes in.

If we could develop efficient ways to use limiting computations to develop programs from examples, we might be able to use them to automate part of the work now done by programmers. We might also be able to use them to do other jobs, such as developing theories from examples (which I take to be part of the work of scientists) and diagnosing ailments from their symptoms (which I take to be part of the work of physicians).

Limiting-computing algorithms might be able to determine what their users have in mind from how they are behaving. Thus, a teaching program might try to figure out how a particular student is thinking to figure out what to do next. A computer interface might use similar methods to try to understand why a user is "stuck."

With such methods, computers might be able to understand us and take over many of the nonclerical jobs that have eluded automation—jobs often said to require "intelligence" [5].

Conclusion

Recall that when Alan Turing developed what we now call the Turing machine in the 1930s [3]—the theoretical precursor of today's digital computer—he was trying to characterize the mental machinery of what were, in his day, called "computers." Not the machines. They hadn't been invented yet. In Turing's day, computers were human clerks who carried out routine calculations to produce values for mathematical tables or accounting ledgers—operations that were not viewed at the time as requiring much intelligence.

What about the capabilities of the people who dreamed up the routines (or programs) the clerks had to carry out? Although computers might not originally have been designed to automate their capabilities, we may be in luck. Computers may have all the machinery needed for that purpose. To get them to do what the more-than-clerks did, we may only have to let them use the machinery they already have—but differently.

As Turing [9] pointed out in 1947, soon after he helped invent electronic computers, "the intention in constructing (computers) in the first instance, is to treat them as slaves, giving them only jobs which have been thought out in detail. Up till the present machines have only been used in this way." Then he asked "But is it necessary that they should always be used in such a manner?" Here, I have tried to answer his question. It isn't, if we are willing to allow them to carry out limiting computations.

Perhaps it is only coincidence, but allowing ourselves to use limiting computations seems to extend the tools available to programmers in much the same way a quite similar limiting process extended the thinking of mathematicians in a way that made calculus, and hence modern science, possible. Who knows what, if anything, allowing programmers to think in terms of limiting-computable algorithms will do.

Let's try to find out. C

References

1. Burgin, M. *Super-Recursive Algorithms (Monographs in Computer Science).* Springer-Verlag, New York, 2005.
2. Gold, E. Limiting recursion. *Journal of Symbolic Logic 30,* 5 (1965), 28–48.
3. Hodges, A. *Alan Turing: The Enigma.* Simon & Schuster, New York, 1983.
4. Kugel, P. Toward a theory of intelligence. *Theoretical Computer Science 317,* 1–3 (2004), 13–30.
5. Kugel, P. Intelligence requires more than computing ... and Turing said so. *Minds and Machines 12,* 4 (2002), 563–579.
6. Kugel, P. Induction, pure and simple. *Information and Control 35,* 4 (1977), 276–336.
7. Popper, K. *The Logic of Scientific Discovery.* Hutchinson, London, 1959.
8. Putnam, H. Trial and error predicates and the solution of a problem of Mostowski. *Journal of Symbolic Logic 20,* 1 (1965), 49–57.
9. Turing, A. Lecture to the London Mathematical Society on 20 February 1947. In *A.M. Turing's ACE Report and Other Papers,* B. Carpenter and R. Doran, Eds. MIT Press, Cambridge, MA, 1986.
10. Turing, A. On computable numbers, with an application to the Entscheidungs problem. *Proceedings of the London Mathematical Society, Series 2,* 42 (1936), 232–265.

Peter Kugel (kugel@bc.edu) is a research associate professor of computer science at Boston College, Chestnut Hill, MA, and a member of the Harvard Institute for Learning in Retirement, Cambridge, MA.