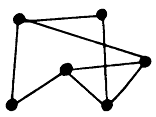
Holographic Proofs

Keeping computers and mathematicians honest

By IVARS PETERSON



After years of scrupulous reasoning and painstaking analysis, Peter Fermax of Enormous State University truly believed he had proved the infamous Snark Conjecture. Now he faced the formidable task of persuading his fellow mathematicians that his 1,210-page manuscript contained not a single error that would invalidate the proof.

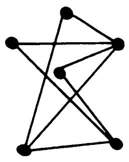
Who would have the time and patience to check each line and certify the proof's integrity?

Unlike his mathematical forebears, who typically had to struggle mightily to win acceptance of their lengthy proofs, Fermax had a tool at his disposal that greatly eased a proof checker's burden. Having already expressed his long chain of reasoning in an acceptable, logical form, he could call upon a supercomputer to convert his statements automatically into a "holographic" version of the same proof.

As in a laser-generated hologram, which makes possible the reconstruction of a scene not only from the whole recorded image but also from each tiny subsection of the image, every statement in a holographic proof contains information about the entire proof. Thus, an error in even a single line of the original proof has a high probability of showing up in any given line of the holographic proof.

In effect, converting a proof into its holographic form greatly amplifies any mistakes present and makes them more readily detectable. To ascertain a proof's validity, a checker has merely to examine, say, a few dozen statements. There's no need to look at the whole proof — no matter how long.

Confident that his proof would withstand scrutiny, Fermax submitted the holographic version to the editors of an electronic journal. Using a rudimentary desktop computer, an anonymous reviewer quickly checked the proof, and the Snark Conjecture became the Fermax Theorem.



Although such a scenario presently exists more in the realm of fantasy than in the real world, computer scientists have over the last few years established the basic principles underlying this sort of scheme. Recent advances in theoretical computer science already point to a remarkably powerful means of checking the correctness of specific answers supplied by a computer.

Quite unexpectedly, the same advances also provide important insights into the strict limits that researchers face when attempting to compute even approximate answers to a variety of problems in computer science. They now know that if it takes an unrealistically long time to compute the exact solutions of certain problems that involve choosing optimal strategies from a host of possibilities, then finding acceptable *approximate* answers will encounter the same barrier.

"This is something that follows in the best traditions of mathematics, where the beauty and the power of a new method comes from connecting things that don't resemble one another," says computer scientist Laszlo Babai of the University of Chicago.

"These are really crucial developments," agrees Leonid Levin of Boston University.

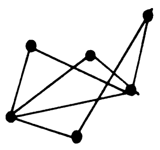
Babai and Levin were among the professors and students at a number of institutions who contributed to the chain of results that led to this convergence between proof checking and the complexity, or difficulty, of computing answers to certain problems.

Moreover, once computer scientists hurdle some of the obstacles in these theoretical results, "I think they may have dramatic practical consequences," Levin says.

At the root of these developments lies the startling notion of a probabilistic, interactive proof. Unlike traditional methods — familiar to any student who has tried to prove one of Euclid's geometrical theorems by constructing a chain of statements inexorably leading to the necessary conclusion — this new technique relies on randomness and the interplay between a "prover" and a "checker" to achieve a practically unassailable proof.

"What is new and very different is that one can convince somebody [of a proof's validity] by using coin flipping and interaction," says Manuel Blum of the University of California, Berkeley, who played a leading role in developing this concept. "I'm excited about that because I think theoretical computer science has introduced a new paradigm here — a paradigm proving to be very powerful for many different reasons and purposes."

Computer scientists several years ago used this concept as the basis for the wonderfully counterintuitive notion of a "zero-knowledge proof." Such a scheme permits someone to persuade others that a particular theorem she has proved is true without giving away anything about how to go about proving the theorem itself (SN: 8/30/86, p.140).



Blum has applied the concept of interactive proofs to what he terms "result checking" or "program checking."

Computer scientists normally count on two methods for ensuring the reliability of computer programs. They can mathematically prove that a given program works correctly for all possible inputs, but that's usually very difficult, if not infeasible. Alternatively, they can test the program by feeding it various inputs, but in this case there's no assurance that the tests cover all the types of inputs a program may encounter.

In contrast, result checking provides a way of determining whether a program, given a certain input, produces the correct answer for that particular input. Conceptually, the process isn't very different from the high-school exercise of substituting a calculated answer back into the original equation to check that the answer is correct.

"The principal idea here is that it's possible to get a program to convince you that the answer it gave you is correct," Blum says. "What you're interested in is either knowing that there's an error somewhere, in which case you would debug the program, or getting strong, convincing evidence that the particular answer you got from the program is correct, even though the program itself may contain bugs."

Blum illustrates his method with the example of determining whether two apparently dissimilar graphs — sets of points, or nodes, connected to one another by lines to form networks — are really the same. One such graph may represent, for example, the electrical properties of a circuit and another the related pattern of connections fabricated in a silicon chip.

Using a standard, off-the-shelf computer program, an engineer can ask whether the two graphs are really the same. But is the "yes" or "no" answer computed by the program correct?

If the two graphs really are the same, a checker has a standard, well-known method for confirming a "yes" answer. If the answer is "no," then the checker can trick an erroneous program into revealing that it has no good reason for its wrong answer. The checker feeds the program the first of the two graphs and either a new version of this graph – in which the nodes are randomly relabeled but the connections remain unchanged – or a randomly relabeled version of the second graph.

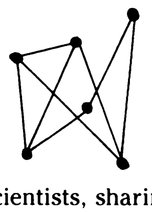
If the program (and everything else, including the computer hardware) is functioning correctly, then it should give a "yes" answer whenever it sees the first graph paired with the relabeled version of the same graph and a "no" answer whenever it sees the first graph and a relabeled version of the second graph. Any other responses indicate that a fault lurks somewhere in the software or hardware.

In this case, because a checker presents one or the other of these alternative pairings randomly and repeats the process a few dozen times, the program has a negligible chance of "guessing" correctly and supplying the appropriate answer each and every time.

"There is a chance that the program can fool this checker, but the chances of doing so are extremely small," Blum says.

On a small scale, Blum already uses such a scheme for routinely checking the output of a calculator program designed to perform certain kinds of arithmetic useful in cryptography. On one occasion, long after he had forgotten his software checker was there working in the background, Blum heard a signal indicating the checker had found a problem.

Although the program handled many pairs of 10-digit numbers successfully, there was a particular pair on which it failed. It turned out that the person who had written the original calculator program had inadvertently introduced a bug that became evident only under very special circumstances.



Of course, Blum's idea didn't develop in isolation. Students contributed to its elucidation, and other computer scientists, sharing insights and communicating results via electronic mail, explored related questions. One such issue concerned how quickly program checking and similar procedures could be accomplished in comparison with the time a computer takes to perform the original calculations.

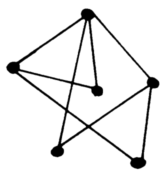
About two and a half years ago, Babai, Lance Fortnow and Carsten Lund, then a student at Chicago but now at AT&T Bell Laboratories in Murray Hill, N.J., established that although a simple personal computer couldn't keep up with a supercomputer's lengthy, convoluted calculations, it could nonetheless interrogate a pair of such supercomputers independently working on the same problem and, within a reasonable time, determine whether the answers were correct.

"There's a mathematical way of actually conducting a session so that [the supercomputers] can be tricked into contradicting one another [if there's an error]," Babai says.

Babai, Levin, Fortnow and Mario Szegedy, also now at Bell Labs, then improved on this procedure by devising a method for transforming a calculation or a mathematical proof into a so-called transparent, or holographic, form that magnifies any errors in the original. This technique involves writing the initial proof as a series of formal mathematical statements and "adding" them together in a special way. Mistakes in any of the original statements would show up in most of the resulting sums.

A checker using a modest, desktop computer could verify the correctness of the transformed calculation or proof by examining only a relatively small number of these sums. "Using a small but reliable machine, we are verifying computation on unreliable hardware with unreliable software and verifying it in a time that is much, much shorter than it takes to do the actual computation," Babai remarks.

"We are not able to prove theorems. We are only able to check proofs," he adds. "This is an important distinction. To come up with a proof, you need ingenuity. To check a proof, you need only a machine."



Other refinements quickly followed, but the most surprising outcomes – and perhaps those of most immediate practical value – lay in the illumination of a deep link between proof checking and the difficulty of solving certain kinds of problems in computer science. In such cases, experience strongly suggests that any conceivable recipe, or algorithm, for finding the solution takes such a long time to run that the exact answer can't be reached within a reasonable amount of time when the number of choices grows large.

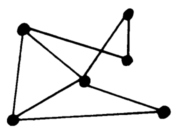
"As far as we can tell, the solution of reasonably sized problems – for example, graphs with a few thousand nodes – would take vastly more than the lifetime of the universe," Babai asserts. This holds even if one could use an ultracomputer, which would have at its disposal all the atoms of the known universe and could perform an operation in the time it takes light to cross an atom.

The "clique" problem, in which one has to find – given a long list of people and their friendships – the largest group of people in which everyone in the group likes everyone else in the group, represents a typical example. Computer scientists have proved that a variety of other important problems that involve searching for optimal strategies in the face of a large number of choices or possibilities have the same level of difficulty.

Formulated by Lund, Szegedy, Rajeev Motwani of Stanford University, and Sanjeev Arora and Madhu Sudan, both students at Berkeley, the latest result proves that for a significant group of such hard optimization problems, one cannot guarantee finding even an approximate answer within a reasonable time. For these cases, it's no easier to find approximate solutions than to find the exact answers.

"This [result] doesn't help you do anything. It doesn't give you a new algorithm to do something faster," says David S. Johnson of Bell Labs. "It helps you by telling you what you can do and what you cannot expect to do."

"What's exciting is that you get [insights] relevant to real-world problems... out of highly theoretical, flights-of-fancy considerations," he adds.



After the breathtaking ride of recent years from one surprising insight to another, computer scientists still face a host of unanswered questions. The latest results on the difficulty of efficiently arriving at approximate solutions apply to only a portion of the hard optimization problems that computer scientists face in handling everything from airline schedules to telephone networks. It still leaves the feasibility and efficiency of many approaches unresolved.

At the same time, computer scientists are exploring what it would take to convert properly expressed mathematical proofs or computations into a transparent, or holographic, form. One difficulty stems from the great length of the transparent proof that one would typically obtain from a given formalized proof.

"We are trying to reduce that size, and once we reduce it to a reasonable level, one would hope that practical applications could be found," Babai says.

Anyone hoping to apply this technique to a real mathematical proof, written initially in a human language, would have to find a way of translating it into a formal language based explicitly on the rules of logic. Only then could the proof be transformed into its transparent guise. At the same time, checking that a large computer system is doing what it's supposed to do is limited by the difficulty of accurately specifying a program's or computer system's objectives.

More immediate practical benefits may lie in the application of Blum's program-checking schemes to computer calculations. With such methods, even in bug-infested computer systems, one can obtain an ironclad guarantee that particular answers are really correct. □