Can Computers Routinely Discover Mathematical Proofs?

STEPHEN A. COOK

Professor of Computer Science, University of Toronto

Thesis: In general, it is much harder to *find* a solution to a problem than to *recognize* one when it is presented.

A simple example of this phenomenon is found in Rubik's cube. Given a scrambled cube, the problem is to arrange each face to have a solid color, by successive moves consisting of twisting the faces. Once the cube has been unscrambled this fact can be verified at a glance. But the process of unscrambling can take weeks for someone ignorant of the cube, even if he is an intelligent problem solver.

A more fundamental example comes from mathematical proofs. Every mathematician knows that it is much easier to verify (or poke holes in) somebody else's proof than to find a proof himself.

Our goal, then, is to formalize the above thesis as a mathematical assertion, and prove or disprove it.

As a guide to how this might be done, let us note that symbolic logic, especially the predicate calculus, provides us with the tools for formalizing mathematical propositions and proofs. Consider the famous open question known as Goldbach's Conjecture as an example. This states that every even integer greater than two is the sum of two primes. It can be formalized in predicate calculus by the following formula:

$$\forall n((n > 2 \wedge 2|n) \supset \exists r \exists s(R(r) \wedge R(s) \wedge n = r + s))$$

Here the symbol $\forall$ stands for "for all," the symbol $\exists$ stands for "there exists," the predicate letter R represents "is a prime," lower case letters stand for integers, and other symbols receive their normal algebraic meaning. In fact, any simple assertion about integers can be represented by a "string" of symbols in this way. (A "string" of symbols just means a finite sequence of symbols written consecutively. The symbols are taken from some fixed finite alphabet.)

In general, to formalize a branch of mathematics one must agree on what objects are to be talked about and what axioms are satisfied. Once this is done a formal theory can be defined. The most general such theory is set theory. In fact the language of set theory is so powerful and the standard Zermelo-Fraenkel axioms for set theory are so strong that most of mathematics can be formalized in this one theory.

The formal theory includes formal assertions (or *formulas*) and formal proofs. A formal proof V of a formula A consists of a finite sequence $B_1, B_2, \ldots, B_n$, A of formulas ending in A, which satisfies certain conditions. Thus the relation "V proves A" is precisely defined, when V and A are each finite strings of symbols. A formula which has a formal proof is called a *theorem.*

The relation "V proves A" has a very important property: A computer can be programmed to rapidly determine, given strings V and A as input, whether the relation "V proves A" holds. This checking action cor-

responds to the *recognition* of a solution to the problem of finding V given A mentioned in our thesis. The question of *finding* a solution in this case has the following interesting instance.

Question: Can one build a computer which, given an assertion A, finds a proof V of A in a reasonable amount of time, if such a V exists of length at most 1,000 symbols.

Certainly no one knows how to build such a computer for general formal theories (such as number theory and set theory). In fact, most mathematicians would probably be quite surprized or (even threatened) if such a computer were found, since it is not uncommon for an interesting and difficult open question in mathematics to turn out to have a solution as short as 1,000 symbols. One thing is certain, namely the technique known as "blind search" will not work. Blind search consists of trying all possible strings V of length at most 1,000 symbols and checking in each case whether V proves A. Since there are 20^{1000} possibilities for V (assuming a 20 symbol alphabet) the result would take any conceivable computer far longer than the projected life time of the sun.

But the failure of blind search is never a strong indication that the problem is intractable. Returning to our example of Rubik's cube, there are approximately 4.3×10^{19} possible patterns for a scrambled cube. Doing blind search through even this relatively small number of patterns is still beyond the power of present day computers, and yet there are children who can routinely put a scrambled cube right in less than a minute. Thus our Question above still stands.

In order to formalize our thesis in general, it is necessary to generalize the notion of formal proof so that it can apply even when the object being proved is not necessarily a mathematical assertion. To this end, let S be any fixed set of strings. (It will still be helpful to think of S as a set of true mathematical formulas).

Definition: An *abstract proof system* for S is an (abstract) relation y *proves* x (where x and y are strings) such that

(1) x is in S iff y proves x, for some string y, and
(2) some computer can recognize the relation *proves* rapidly.

Condition (2) must be explained further. First of all, by a *computer* we mean a suitable mathematical model of a computer, such as a Turing machine. (The definition of Turing machine is not important for our present purpose.) Second, the computer is said to *recognize* the relation "proves" if, when it is presented with an arbitrary pair (x, y) of strings as input, it halts with a "yes" or "no" output according to whether or not y proves x. Finally the recognition is *rapid* if the number of steps taken by the computer to produce the output is bounded by a fixed power of the combined length n of x and y (for example, it suffices if the number of steps is n^2 or n^3).

Note that formal theories are examples of abstract proof systems, and below we shall give other examples.

Definition: The abstract proof system is *short* if there is a fixed exponent k such that whenever some string y exists in condition (1) above, the string y can be chosen to be short, in the sense that (length (y) $\leq$ (length (x))^k.

Now we are in position to define an important class of sets, called NP.

Definition: A set S of strings is in the class NP iff S has a short abstract proof system.

Here are some examples of sets in NP.

(1) Fix any formal theory, such as formal set theory. Note that the set of theorems of the theory is in general not in NP, because the theorems do not necessarily have short proofs. However, we can define the set THM to consist of all strings of the form

$A\#\underbrace{11\cdots1}_{n}$, where A is a theorem with a formal proof in the theory of length at most n. Then the set THM is in NP, since the formal proofs in the theory can naturally be made to provide a short abstract proof system for THM.

(2) The set SAT consists of all satisfiable propositional formulas; that is formulas such as $(\neg P \wedge Q) \vee (Q \wedge \neg R)$ consisting of propositional atoms (such as P, Q, R), the propositional connectives $\wedge$ (and), $\vee$ (or), and $\neg$ (not), and parentheses. Such a formula is satisfiable if there is an assignment of either "true" or "false" to each propositional atom which makes the whole formula true. The set SAT is in NP, since we can define a short abstract proof of a satisfiable formula to consist simply of a truth assignment which makes the formula true. (It is easy to check whether a given truth assignment works.)

(3) Many problems from operations research can be coded as sets in NP. One famous example is the so-called traveling salesman problem: Given a set of cities (i.e., points on the plane) together with the (approximate) distance between every pair of cities and a trial distance D, determine whether there is a path (or tour) of length at most D which starts and ends at one city and hits all other cities. The set of all such problem instances for which such a tour exists is known as TSP. The set TSP is in NP, since an abstract proof for a problem instance can consist of a tour of length at most D. It is easy to check whether a given list of cities constitutes a tour, and easy to compute the length of the tour.

All of the above three problems are in the class NP, but there is no known computer program for solving any of them in a reasonable amount of time for moderately large inputs.

Any set S in the class NP, including the above three examples, can be regarded as a problem. An instance of the problem is a string x, and a solution to the problem x is a string y such that y proves x in the abstract proof system corresponding to S. If x is not in S, then there is no proof y, and the problem x has no solution. The crucial property of a problem S in NP is that a solution y to an instance x can be recognized rapidly once it is found. However, we do not in general know how to find the solution rapidly, even when it exists. If a solution can always be found rapidly when it exists, then the problem is in the class P. More generally:

Definition: A set S of strings is in the class P iff some computer can recognize S rapidly.

The notions of "computer" and "recognize S rapidly" are defined in the same way as in the definition of abstract proof system.

The class P of sets is trivially a subclass of NP, but is it a proper subclass? In fact, we are now in a position to formalize our thesis as follows:

Formal thesis: $P \neq NP$

In other words, there is a problem S in NP (i.e. the solutions to instances are easy to recognize) which is not in P (the solutions are hard to find). In fact, if there is one such problem S it follows that there are many common such problems, as we shall see.

The question P = NP? is a precise and important mathematical problem which remains open, despite the best efforts of many mathematicians for the past dozen years. Substantial interest in the question was generated about twelve years ago when the present author (1) and R. M. Karp (2) developed a notion now called *NP completeness.* If a set S is NP complete, it has the following two properties:

(1) S is in NP.
(2) If S is in P, then P = NP.

In fact, property (2) can be strengthened to say that given any problem T in NP, any fast algorithm for S can be transformed into a fast algorithm for T.

What is surprising about this notion is that

so many important computational problems turn out to be NP complete. Not only are the three examples THM, SAT, and TSP above all NP complete, but the textbook (3) by Garey and Johnston lists over 300 others. Thus if our formal hypothesis P ≠ NP holds, then none of these many problems is in NP.

Showing that a problem S is NP complete has a significant practical consequence. Suppose you would very much like to find a fast algorithm for solving S, but all your best efforts have failed. If you can show S is NP complete, then you have a good excuse for failure, since success would imply P = NP. In particular, a fast algorithm for S would provide a fast algorithm for each of the 300 NP complete problems in (3).

I, for one, am hoping someone will prove that P ≠ NP in the next decade or two. If a proof is found, I believe the result will rank with the Gödel Incompleteness Theorem as an interesting mathematical theorem with important philosophical consequences.

References

1. Cook, S. A. The complexity of theorem proving procedures. *Proceedings. 3rd ACM Symp. on Theory of Computing (1971).* 151–158.
2. Karp, R. M. Reproducibility among combinatorial problems, in: *Complexity of Computer Computations,* R. E. Miller and J. W. Thatcher, eds. Plenum Press, New York, 1972, 85–104.
3. Garey, M. R. and D. S. Johnson, Computers and Intractability: *A Guide to the Theory of NP-Completeness,* W. H. Freeman, San Francisco, 1979.