Daniel Delany
Theory of Computation
5/5/2008

Classifications in Computational Complexity Theory

In the theory of computation, one often analyzes "computational problems," that is, questions with correct answers that we may wish a computer to solve. The subfield which examines the computational resources required by these problems (time and memory) and classifies these problems into sets which have similar characteristics is called "computational complexity theory." While classifying problems into sets, many of which are computationally intractable (or even impossible), may seem like an exercise in futility, we can show that this method of definition establishes inherent mathematical characteristics about the problems which may help us understand which of them are worth attempting to find an algorithmic solution to, and which of them are likely to remain intractable. In the following pages, I'll describe some of these characteristics, and highlight the most important of these sets of problems. Below is a diagram which displays a small part of the computational complexity hierarchy (the largest circle represents the set of all computational problems); we will use this diagram as a roadmap to understanding the basics of complexity theory:

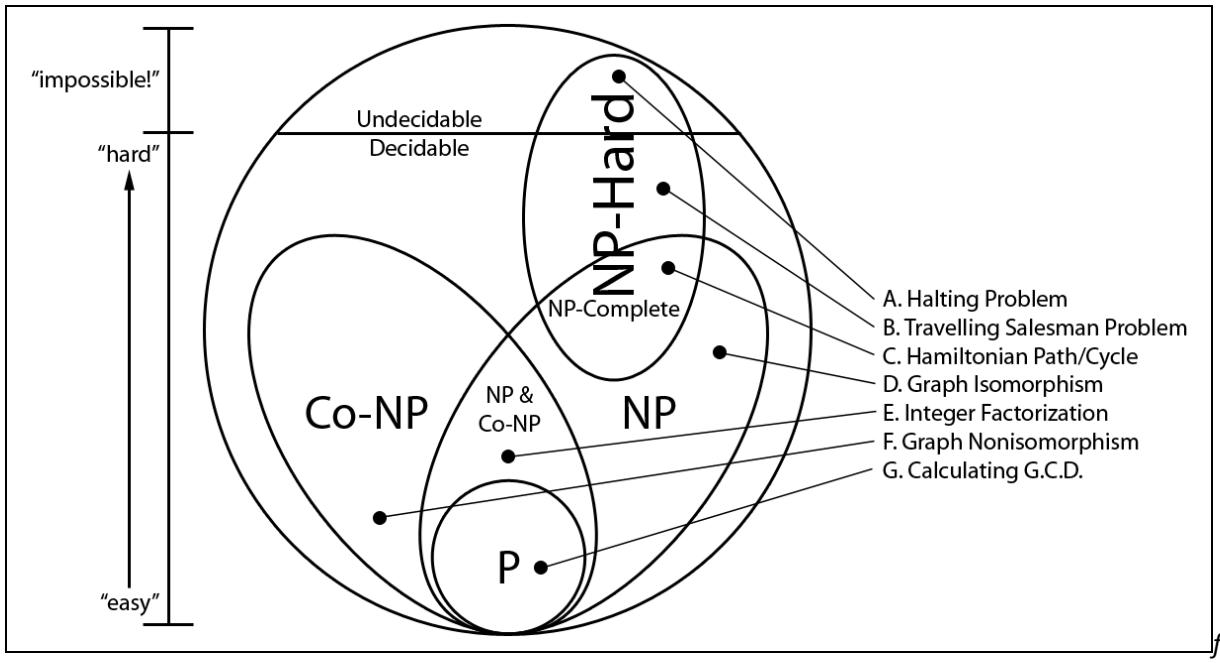

fig. 1: A few important sets of problems in complexity theory

Let's begin by looking at the main subdivision within computational problems – decidability vs. undecidability. In general, a decidable problem is one for which an algorithmic solution which will always produce a correct answer in finite time for any valid input exists. Conversely, an undecidable problem is one for which no such algorithm exists. While there are a multitude of intuitively decidable problems, proving the undecidability of a problem is much more difficult. Let us examine one such problem: the halting problem (see fig. 1A). This problem addresses the issue of the existence of an algorithm to

determine whether a given Turing Machine program i, with given input j, will ever stop executing and return a result. It is important to note that the question of decidability rests not on the existence of an algorithm which can test a single case, or even a set of cases; rather, it must be able to take *any* given input and produce the desired result. For example, it would be relatively easy to construct a pseudo-halting-problem algorithm which detects the existence of a set of infinite loops written in the following format:

```
int a = 1;
while(a) { noOp(); }
return;
```

From this example alone, it may appear that the halting problem is intuitively easy: just look for infinite loops in a program. However, a general solution must be able to solve the halting problem for *any* given program, which is significantly harder. For example, imagine another program which tests Goldbach's Conjecture, which states that every even integer greater than 2 can be written as the sum of two primes:

```
int a = 4;
while(1) {
   if(noTwoPrimesSumTo(a)) { return; }
   else { a += 2; }
}
```

With this example, we've showed that solving the general halting problem is *at least as hard* as proving Goldbach's conjecture, a feat which has eluded human mathematicians for more than two-and-a-half centuries! Below, we'll prove that it's probably *harder* than that.

The nonexistence of a solution to the halting problem can be proved using Cantor's diagonalization technique, which I'll briefly describe: Imagine a matrix, M(i,j), for which the rows represent all possible Turing Machine programs, and the columns represent all possible inputs to these machines. Assuming a solution to the halting problem exists, we can run it on all Turing Machines for all possible inputs, placing a 1 in the matrix whenever a machine halts for a given input, and a 0 whenever a machine does not halt. Each matrix row, then, represesents a set of halting algorithm outputs for a given TM for all possible outputs. However, if this is the case, let us examine the inverse of the diagonal of this matrix: this sequence, itself, can be said to represent a set of halting problem outputs for a TM. However, the inverse of the diagonal is a sequence which, inherently, is not a row in M. Since the rows in M are said to represent the halting problem outputs of ALL possible TM's, but we also have a TM halting output sequence which is not a row in M, a contradiction exists, and we have proven that our initial assumption (the existence of a solution to the halting problem) is false.

Let us now move from the realm of undecidable problems to those which are decidable, that is, problems for which an algorithm can be formulated which always returns a correct answer in finite (though not necessarily practical) time. A simple example of one such problem is the problem of finding an algorithm to determine the greatest common divisor (GCD) of any two integers. Such an algorithm can be represented tersely as:

```
int gcd(int a, int b) { return ( b != 0 ? gcd(b, a % b) : a ); }
```

It is fairly easy to prove, though the proof is not shown here, that this algorithm will produce the correct answer for any integers a and b. Actually, this problem is not only decidable, but falls into a very special, small subset of the decidable set known as P, which represents all problems that can be decided by deterministic Turing machines in polynomial time (see fig. 1G) or $O(n^k)$ in big-O notation. Specifically, the GCD problem is $O(n^2)$, since it can be easily shown that, in the worst case, the algorithm will require O(n) divisions, each of which has time complexity O(n).

Let's now zoom out to a larger subset of problems within the decidable set: the NP subset (see fig. 1D). NP, or Nondeterministic-Polynomial problems, are decision problems (questions with a “yes” or “no” answer) that can be solved in polynomial time by a non-deterministic Turing Machine. A non-deterministic Turing Machine is a Turing Machine that, unlike normal, deterministic Turing Machines, can move to more than one possible state from any given state and input (that is, the current state and input does not uniquely specify the machine's next state). This is perhaps best intuitively imagined as a deterministic TM that, upon reaching one of these state decisions, copies itself and runs one deterministic TM on each of the possible next states. The NP subset is, itself, a subset of the EXPTIME set (the set of all problems which can be solved by deterministic Turing Machines in exponential [$O(n^n)$] time), since a deterministic TM could be programmed to traverse an entire decision tree in exponential time. From this definition, it is easy to see that the P set discussed previously is a subset of the NP set, since a P-problem can be solved easily with a non-deterministic Turing Machine that never branches (that is, a non-deterministic TM that acts like a deterministic TM).

An alternate, though equivalent, definition of the NP set is the set of problems verifiable by a deterministic TM in polynomial time; that is, if given an answer which is claimed to be a solution to an NP-problem, we can verify that the solution is true in polynomial time with a standard TM. This is equivalent to our original definition: if we suppose, as we originally stated, that a non-deterministic TM can solve a problem in polynomial time, then it does so by trying all possible paths and returning the correct path, a path which can be verified deterministically by a TM performing only the supposed “correct” state transitions. Conversely, if we suppose that a deterministic polynomial verifier exists, our first definition follows since a non-deterministic TM could test the verifier on all possible solutions and return the solution which passes the test in polynomial time.

One important example of a problem which falls in the NP set is the problem of graph isomorphism; that is, given two graphs, are the two identical? Answering this question must be done non-deterministically, since one must check each vertex in one graph for equivalence to each vertex in the other graph. However, once a guess is made, the guess can be verified as correct using a simple, deterministic polynomial algorithm (which will not be described here).

One notable set of problems related to the NP set is the co-NP set (see fig. 1F), the NP set's complement. Problems in the co-NP set have the property that, if a given solution is *not* a correct solution, its incorrectness can be proven in deterministic polynomial time. The graph isomorphism problem is not known to be a co-NP problem, since, while a true isomorphism can be verified in deterministic polynomial time, a deterministic polynomial algorithm proving that a given graph is *not* an

isomorphism is not known to exist. Many problems fall into both the NP and co-NP sets, including the problem of integer factorization. This problem asks, given two integers m and n, if n has a factor less than m and greater than one. The problem must be attacked by trying all possible factors with a non-deterministic TM. However, if an answer is provided, one can quickly use long division to show that such a solution is correct, or the AKS primality test on m's prime factors to prove that such a solution is incorrect. It is believed (though not proven) that the P set is a strict subset of both NP and co-NP; this makes sense intuitively since a solution that can be obtained "easily" should also be both easily verifiable and falsifiable.

Let us now examine a new set of problems known as NP-Hard (see fig. 1A-C). NP-Hard problems are problems which, by definition, are *at least as hard* as the hardest problems in the NP set. This includes some problems in the NP set (known as NP-complete, discussed later), some problems outside the NP set but still in the decidable set, and problems that are undecidable. It also includes search problems and optimization problems, whereas the NP set is restricted to decision problems. Intuitively, this means that if a deterministic polynomial solution were found for an NP-Hard problem, the same algorithm could be applied to any problem in the NP set (though not any problem in the NP-Hard set) to solve it in deterministic polynomial time as well. Since such an algorithm is purely theoretical, and most mathematicians doubt its existence, it is useful to consider an imaginary "oracle machine." A problem's oracle machine is a device that, using means unknown, solves an otherwise non-deterministic problem in deterministic polynomial time. The definition of the NP-Hard set implies that, if such a machine existed for any particular NP-Hard-problem, the same machine would be able to solve *all* NP-problems in deterministic polynomial time as well, while the reverse is not necessarily true.

An example of an undecidable NP-Hard problem is the halting problem discussed earlier (see fig. 1A). This can be proven informally by showing that any NP-problem (including all NP-Complete problems) can be written algorithmically in such a way that the algorithm returns (halts) if the answer to the decision problem is "yes", and stays in an infinite loop if the answer is "no". If a solution to the halting problem exists, then the question of whether or not the aforementioned algorithm ever returns can be answered in deterministic polynomial time, effectively solving the NP-problem in the same deterministic polynomial time. Attacking these sorts of problems (undecidable NP-Hard) in hopes of finding general-NP solutions, however, is fruitless, since we can prove that undecidable problems *cannot* be solved in finite time. It is believed, though difficult to show mathematically, that all undecidable problems are in the NP-Hard set, since a problem which cannot be solved in finite time is inherently harder than one which can (ie. all NP problems).

There also exist NP-Hard problems which are decidable, but are not in NP (that is, are not NP-Complete) (see fig. 1B). An example of this is the Travelling Salesman Problem, which asks, given a list of cities and their coordinates, for the shortest route which visits all of the cities. First of all, it is obvious that the problem is not in NP, since it is not a decision problem, a requirement of the NP set. Additionally, we can informally show that, even if the problem is solved, verifying the solution could not be done in deterministic polynomial time, another requirement of NP problems: given a supposed solution, the only way to determine whether or not the route is actually the shortest is to try all possible routes and test to see if they are, indeed, longer than the solution route – this is, in itself, an NP-

problem! However, since solving the Travelling Salesman Problem involves trying all possible paths, it is reducible to all NP-problems, which means that a deterministic polynomial solution to the TSP would solve all NP-problems, which means that the TSP is, indeed, NP-Hard.

Finally, we come to one of the most important subsets in complexity theory: the NP-Complete set (see fig. 1C). This set represents the intersection between the NP set and the NP-Hard set, and all problems in this set have all characteristics of both NP and NP-hard (that is, they are decision problems which can be solved by non-deterministic Turing Machines in polynomial time, they have solutions which can be verified in polynomial time by deterministic TM's, and they are at least as hard as any other problem in NP). An example of an NP-complete problem is the problem of whether or not, in a graph, a Hamiltonian path exists. A Hamiltonian path is an undirected graph which visits each graph vertex exactly once. A related, similarly NP-complete problem is the question of whether or not a Hamiltonian *cycle*, that is, an undirected graph which visits every vertex at least once and returns to the starting vertex, exists. The fact that these are in the NP set is easily proved informally: determining whether or not a Hamiltonian path or cycle exists in the first place requires testing all possible graph edges non-deterministically. However, once such a path or cycle is found, showing that it is indeed a path or cycle is trivial: all we have to do is ensure that each vertex is visited exactly once! Moreover, it can be shown that this problem is not *just* NP, it is NP-complete, meaning that it is as hard as any NP problem, and a polynomial solution to it would solve all NP problems.

These problems are particularly interesting to mathematicians and computer scientists because, if a single one of them were shown to be deterministically solvable in polynomial time (as many problems previously assumed to be NP problems have), it would prove that all NP-problems are also deterministically solvable in polynomial time. While this is a true statement of any NP-Hard problem, NP-complete problems are more appealing to mathematicians, as they may be fundamentally easier to reduce to P-problems than other NP-hard problems. Proving this would prove that the NP set and the P set are actually equal to one another. While the majority of mathematicians and theoreticians believe this is not the case, proving it one way or the other remains one of the largest problems in computer science today. The ability to perform NP operations in polynomial time would fundamentally change the landscape of the computer world forever.

The existence of problems which, if solved, would solve all NP problems, seems like a mathematical rarety, if not a statistical impossibility. However, it's relatively easy to informally prove the existence of this sort of problem: Imagine we have a deterministic Turing machine M which is guaranteed to halt in polynomial time, and we want to know if there exists a polynomial-sized input that M will accept. This problem is not only in NP, but we can prove that all NP problems are reducible to this problem. Why? This is the very definition of an NP problem: a problem which can be verified by a deterministic Turing machine in polynomial time! While this example may be trivial, NP-Complete problems are not interesting because they exist – we can create a formal definition of almost any machine we can imagine. They are interesting because a number of physical processes that are relevant to “life in the real world,” processes that we would very much like to be able to simulate efficiently, fall into the NP-complete category. The game of Tetris is NP-complete, as is the problem of packing items of different volumes into a finite number of limited capacity slots, or bins.

Given a problem Π, proving NP-Completeness requires two things:

1. Prove Π is in NP
2. Prove that, for every problem Π' in NP, Π' is reducible to Π.

However, it would be nearly impossible to test the second requirement, since there are an infinite number of problems in NP. However, we know that, if a problem is already known to be NP-complete, all problems in NP are already reducible to it. Therefore, instead of requiring that we test all problems in NP, we can change the second step to:

2. Find a single NP-Complete problem, Π', that is reducible to Π.

This single piece of knowledge alone makes NP-Completeness far easier to prove, since all we need to do is prove our problem's equivalence with a similar problem which has been proven NP-Complete.

One wonders if any of this complexity theory can be applied to ideas of consciousness, the brain and the prospects for artificial intelligence. After all, this is theoretical math we're dealing with, not hard scientific data about the brain's inner workings. However, complexity theory deals with decision problems, search problems, and optimization problems, tasks humans perform on a daily basis, so I believe a parallel may be drawn. Currently, human minds are not known to solve any NP or NP-complete problems "automatically," that is, without informing the conscious mind. The conscious mind seems to have tricks for approximating solutions to many of these sorts of problems (for example, if I gave you the Travelling Salesman Problem, you'd start at one side of the map and draw out an approximately efficient route to the other), but many, such as integer factorization, cannot be estimated. It will be interesting to see, as scientists continue to study and attempt to understand the inner-workings of the mind, if we can classify the methods the brain uses to solve problems in the same hierarchy we've built to understand computers, our own logical creations.

Does P=NP? More interestingly, has evolution allowed the mind to take advantage of this, if it is the case? Mathematicians have theorized about some of the consequences of polynomial-time NP-problem execution; some believe it would allow computers to constantly generate arbitrary-length "original" and "creative" strings, a distinctly human ability. Steven Rudich, a complexity theorist at Carnegie-Mellon University, once conceptualized the difference between P and NP sets by saying "I can recognize great music, but I can't create great music," implying that, while P lets us verify solutions easily, NP implies that creating solutions is much harder. This statement was surely meant as a conceptual metaphor, but it raises some truly interesting questions: if we are able to prove P=NP, does this allow us to create computational constructs that can be creative as easily as they can recognize creativity? Is it possible that some of humanity's biggest questions and creations can be explained by polynomial solutions to NP problems? As we continue to crack the puzzles surrounding the ways in which the brain physically manifests consciousness, it will be quite interesting to see if we can discern the complexity set our brain most often operates inside of, and whether or not this same complexity set can be equaled, or even improved upon, by the computers of tomorrow.

Sources Cited

Salavatipour, Mohammad R. “CMPUT 204: Algorithms I – Lecture 33.” Fall 2007. University of Alberta. http://www.cs.ualberta.ca/~mreza/courses/204-F04/lectures/lecture33.pdf

Aaronson, Scott. “PHYS771 – Quantum Computing Since Democritus.” Fall 2006. University of Waterloo. http://www.scottaaronson.com/democritus/lec6.html

Fortnow, Lance and Gasarch, Bill. “P=NP and the Arts.” March 24, 2005. http://weblog.fortnow.com/2005/03/pnp-and-arts.html

“Complexity Theory.” Wolfram MathWorld. http://mathworld.wolfram.com/ComplexityTheory.html

Keuneke, Anne. “Design and Analysis of Algorithms - NP-Hard and Decision Problems.” California State University, Chico. http://www.ecst.csuchico.edu/~amk/foo/csci356/notes/ch11/NP5.html

Billings, John. “Diagonalisation proof of the undecidability of the Halting Problem.” January 26, 2006. University of Cambridge. http://www.cl.cam.ac.uk/~jnb26/halting_problem.pdf