6.080 / 6.089 Great Ideas in Theoretical Computer Science
Spring 2008

Scribe Notes: Introduction to Computational Complexity

Jason Furtado

February 28, 2008

1 Motivating Complexity Theory

Penrose's Argument (continued)

Last lecture, Roger Penrose's argument was introduced. It says that it was impossible for a computer to simulate the human brain. His argument is based on Godel's Incompleteness Theorem. Penrose's argument says that, within any formal system F, we can construct a Godel sentence $G(F)$ that the computer cannot prove but a human can prove it.

Professor Aaronson proposes that there is a simple way to see that there must be a problem with Penrose's Argument or any similar argument. Penrose is attempting to show that no computer could pass the Turing Test. The Turing Test states that a computer is intelligent if it could carry on a textual conversation with a human and the human would be unable to tell that he or she was having a conversation with a computer rather than another human. A Turing Test lasts a finite amount of time (it is a conversation). If you could type incredibly quickly, say 5000 characters per minute, the number of possible instant messaging conversation you could have is finite. Therefore, in principle, a giant lookup table could be constructed that stored an intelligent human response to every possible question that could be asked. Then the computer could just consult the lookup table whenever a question is asked of it.

The problem with this lookup table is that it would have to be incredibly large. The amount of storage necessary would be many times larger than the size of the observable universe (which has maybe 10^{80} atoms). Therefore, such a system would be infeasible to create. The argument has now changed from theoretical possibility to a question of feasible given resources. The argument against such a system has to do with the amount of memory, processing power, etc. needed to create a system that could reliably pass the Turing Test. Since the system is theoretically possible, the question is whether or not the amount of resources necessary is a reasonable amount.

In order to have such a conversation, we will need a way to measure the efficiency of algorithms. The field that studies such issues is called *computational complexity theory*, and will occupy us for most of the rest of the course. In contrast to what we saw before, computational complexity is a field where almost all of the basic questions are still unanswered – but also where some of what we *do* know is at the forefront of mathematics.

Dijkstra Algorithm

In 1960, the famous computer scientist Edsgar Dijkstra discovered an algorithm that finds the shortest path between two vertices of a graph, in an amount of time linear in the number of edges. The algorithm was named after him (Dijkstra's Algorithm), and something like it is used (for example) in Google Maps and every other piece of software to find directions. There's a famous anecdote where Dijkstra (who worked in a math department) was trying to explain his new result to a colleague. And the colleague said, "but I don't understand. Given any graph, there's at most a finite number of non-intersecting paths. Every finite set has a minimal element, so just enumerate them all and you're done." The colleague's argument does, indeed, show that the shortest path problem is computable. But from a modern perspective, showing a problem is computable does not say much. The important question is whether a problem is solvable *efficiently.*

Efficiency

Computer scientists are interested in finding efficient algorithms to solve problems. Efficiency is typically measured by considering an algorithm's running time as a function of the size of the input. Here *input size* usually (not always) refers to the number of bits necessary to describe the input. This way of describing efficiency avoids being too tied to a particular machine model (Macs, PC's, Turing machines, etc.), and lets us focus on more "intrinsic" properties of the problem and of algorithms for solving it.

2 Circuit Complexity

A good place to begin studying complexity questions is *circuit complexity.* Historically, this is also one of the first places where complexity questions were asked.

Let's pose the following general question:

Given a Boolean function $f : \{0,1\}^n \to \{0,1\}$, what is the size of the smallest circuit that computes it? (That is, how many gates are needed?)

As an example, suppose we're trying to compute the XOR of the n input bits, $x_1, \ldots, x_n$. Then how many gates to do we need? (For simplicity, let's suppose that a two-input XOR gate is available.) Right, $n-1$ gates are certainly sufficient: XOR x_1 and x_2, then XOR the result with x_3, then XOR the result with x_4, etc. Can we show that $n-1$ gates are necessary?

Claim: You need at least $n-1$ gates in order for the single output bit to depend on all n input bits using 2-input gates.

Proof: Using the "Chocolate-breaking" argument as a basis, we can view each input as a group. Each XOR function joins two groups, so you then have one less total group than before. If you continue to join groups until there is only a single group, you would have used $n-1$ XOR functions to join the n inputs.

Note: In lecture, it was brought up that another measure of efficiency in circuits is the *depth* of the circuit. This is the maximum number of gates needed to traverse the circuit from an input to the output. For a XOR circuit of n inputs constructed using two-input XOR gates, one can easily show that a depth of $\log n$ is necessary and sufficient.

Shannon's Counting Argument

Is there a Boolean Function with n inputs that requires a circuit of exponential size in n?

Well, let's look at some possible candidates. Does the Majority function require an exponential-size circuit? No, it doesn't. What about *SAT* and other *NP*-complete problems? Well, that's getting ahead of ourselves, but the short answer is that no one knows!

But in 1949, Claude Shannon, the father of information theory, mathematical cryptography and several other fields (and who we'll meet again later), gave a remarkably simple argument for why such a Boolean function must exist.

First, how many Boolean functions are there on n inputs? Well, you can think of a Boolean function with n inputs as a truth table with 2^n entries, and each entry can be either 0 or 1. This leads to 2^{2^n} possible Boolean functions: not merely an exponential number, but a *double*-exponential one.

On the other hand, how many circuits are there with n inputs and T gates? Assume for simplicity that there are only NAND gates in our circuit. Then each gate will have at most $n+T$ choices for the left input and most $n+T$ choices for the right input. Therefore, there can be no more than $(n+T)^{2T}$ circuits possible. Every circuit can compute only one Boolean function. And therefore, if we want to represent all 2^{2^n} Boolean functions, then we need $(n+T)^{2T} \geq 2^{2^n}$. Taking the log of both sides we can estimate that T is $\frac{2^n}{2n}$. If T were any smaller than this, there would not be enough circuits to account for all Boolean functions.

Shannon's argument is simple yet extremely powerful. If we set $n = 1000$, we can say that almost all Boolean functions with 1000 inputs will need to use at least $2^{1000}/2000$ NAND gates in order to compute. That number is larger than 10^{80}, the estimated number of atoms in the visible universe. Therefore, if every atom in the universe could be used as a gate, there are functions with 1000 inputs that would need a circuit much bigger than the universe to compute.

The amazing thing about Shannon's counting argument is that it proves that such complex functions must exist, yet without giving us a *single specific example* of such a function. Arguments of this kind are called *non-constructive.*

3 Hartmanis-Stearns

So the question remains: can we find any *concrete* problem, one people actually care about, that we can prove has a high computational complexity? For this question, let's switch from the circuit model back to the Turing machine model.

In 1965, Juris Hartmanis and Richard Stearns showed how to construct problems that can take pretty much any desired number of steps for a Turing machine solve. Their result basically started the field of computational complexity, and earned them the 1993 Turing Award. To prove their result (the so-called Hierarchy Theorem), they used a "scaled-down" version of the argument that Turing had originally used to prove the unsolvability of the halting problem.

Let's say we want to exhibit a problem that any Turing machine needs about n^3 steps to solve. The problem will be the following:

Question: Given as input a Turing Machine M, input x and integer n does M halt after at most n^3 steps on input x?

Claim: Any Turing machine to solve this problem needs at least n^3 steps.

Proof: Suppose there were a machine, call it P, that solved the above problem in, say, $n^{2.99}$ steps. Then we could modify P to produce a new machine P', which acts as follows given a Turing machine M and input n:

1. Runs forever if M halts in at most n^3 steps given its own code as input.

2. Halts if M runs for more than n^3 steps given its own code as input.

Notice that P' halts in at most $n^{2.99}$ steps (plus some small overhead) if it halts at all – in any case, in fewer than n^3 steps.

Now run $P'(P', n)$. There is a contradiction! If P' halts, then it will run forever by case 1. If P' runs forever then it will halt by case 2.

The conclusion is that P could not have existed in the first place. In other words, not only is the halting problem unsolvable, but in general, there is no faster way to predict a Turing machine's behavior given a finite number of steps than to run the machine and observe it. This argument works for any "reasonable" runtime (basically, any runtime that's itself computable in a reasonable amount of time), not just n^3.

4 Notation for Complexity

In later lectures, it will be helpful to have some notation for measuring the growth of functions, a notation that ignores the "low-order terms" that vary from one implementation to another. What follows is the standard notation that computer scientists use for this purpose.

Big-O Notation

We say $f(n) = O(g(n))$ if there exist constants a, b such that $f(n) < ag(n) + b$ for all n.

The function $g(n)$ is an upper bound on the growth of $f(n)$. A different way to think of Big-O is that $g(n)$ "grows" at least as quickly as $f(n)$ as n goes to infinity. $f(n) = O(g(n))$ is read as "$f(n)$ belongs to the class of functions that are $O(g(n))$".

Big-Omega Notation

We say $f(n) = \Omega(g(n))$ if there exist $a > 0$ and b such that $f(n) > ag(n) - b$ for all n.

Intuitively, the function $f(n)$ grows at the same rate or more quickly than $g(n)$ as n goes to infinity.

Big-Theta

We say $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ AND $f(n) = \Omega(g(n))$.

Intuitively, the function $f(n)$ grows at the same rate as $g(n)$ as n goes to infinity.

Little-o

We say $f(n) = o(g(n))$ if for all $a > 0$, there exists a b such that $f(n) < ag(n) + b$ for all n.

Intuitively, $f(n)$ grows strictly more slowly than $g(n)$. So $f(n) = O(g(n))$ but not $\Theta(g(n))$.