CSC 5170: Theory of Computational Complexity **Lecture 5**
The Chinese University of Hong Kong 8 February 2010

So far our notion of realistic computation has been completely deterministic: The Turing Machine gets an input, performs some operations that are completely determined by the input, and produces an output. We were interested in how much resources – specifically time and space – were necessary to perform these computations.

Our model of a Turing Machine fails to account for one element that can presumably be found in nature and therefore taken advantage when performing computations: randomness. But the problems we have been looking at so far do not refer to randomness at all. For instance, when we ask if a CNF formula ϕ has a satisfying assignment, the answer is a predetermined "yes" or "no" – no chance involved. What role do we expect randomness to play in such a setting?

Here is a historical example. The *minimum cut* problem asks to find the smallest set of edges that separates a given graph into two components. It has been known that this problem is polynomial-time solvable since the 1950s. However the algorithms for this problem were very slow, and could run for up to $O(n^4)$ steps on a graph with n vertices (on a RAM machine).

In the 1990s David Karger found an extremely simple way to compute the minimum cut in only $\tilde{O}(n^3)$ time by running the following procedure:[1]

```
On input a graph $G$:
    Run the following experiment on $G$ for $n^2 \log n$ times:
        For $i := 1$ to $n - 1$:
            Take a random edge of $G$ and merge its endpoints.
        At this point there are only two "super-vertices" left.
        Remember the cut between the two vertices.
    Return the smallest cut among all the ones found.
```

This procedure is *randomized*: The randomness does not come from the input G but from the algorithm itself. Therefore, for different choices of the randomness, the algorithm will output different cuts. So in what sense can we say that the cut output by the algorithm is the minimum one? This is a remarkable fact that Karger proved: Although for different choices of the randomness the algorithm will output different cuts, and some of them won't be minimum, "most" choices of the randomness will in fact result in the algorithm producing a minimum cut as its output.

Notice the role of randomness played in Karger's algorithm: The randomness itself has nothing to do with the input, but we used it as an external resource that helped us speed up the computation. Its role here is similar to that of a catalyst in a chemical reaction. The catalyst helps the reaction go through more quickly, or at lower temperature, but does not affect the reaction itself.

There are other settings in computation where randomness plays an even more fundamental role, like in communication and cryptography. But today we will focus on the role randomness plays

[1] A variant of the algorithm can be made to work even faster, in time $\tilde{O}(n^2)$.

in speeding up computations. The basic question we will attempt to answer is the following one: Does randomness really help in speeding up computations and if so, to what extent?

1 Randomized computation

To account for the use of randomness we will need to upgrade our basic model of computation - the Turing Machine. If you look at Karger's min-cut algorithm it appears that randomness is an internal resource; at some point, the algorithm just says "do this at random". However, for our purposes it will me more helpful to think of the random choices as being provided to the algorithm externally. This is similar to the way we can view nondeterminism either as an internal resource (the Turing Machine making choices) or an external one (the choices are written ahead of time on a special tape).

A *randomized Turing Machine* contains, in addition to its other tapes, a special *random tape* which is initialized with a sequence of random and uncorrelated bits $r_1, r_2, \dots \sim \{0,1\}$.[2] As soon as we allow the algorithm access to a random tape, its behavior is no longer completely determined by the input x. Now for every x we have a distribution of outputs depending on the how the random tape was initialized. Often it will be convenient to speak of this randomness explicitly, and we can indeed think of a randomized algorithm A as a *deterministic* algorithm that takes both a real input x and a "random" input $r = (r_1, r_2, \dots)$, denoting the setting of the random tape. (This is similar to the way we view a nondeterministic TM as a verifier that receives its witness on the nondeterminism tape.)

An issue that occurs in defining the *running time* of randomized computations is that the running time itself may be a random variable. It might even be that for some settings of the randomness the computation never terminates. However, we will mostly study computations in which the running time is bounded by $t(n)$ for all inputs of length n and all settings of the random tape. In this case we say that the randomized algorithm runs in time $t(n)$, and without loss of generality we can then model the random tape of the algorithm as a string of length $t(n)$.

For decision problems, we will think of a randomized algorithm as "solving" the problem as long as it behaves correctly on *all* inputs but for "most" choices of the randomness. Depending on what we allow the algorithm to do on the remaining, "bad" choices of randomness we obtain three different definitions of randomized complexity classes.

The first option is to require the algorithm to be correct over most choices of the randomness, but to allow it to output an arbitrary answer otherwise. This yields the class of decision problems BPP (for bounded probabilistic polynomial-time).

Definition 1. The class BPP consists of all decision problems L for which there exists a randomized algorithm A running in probabilistic polynomial-time and such that for every $x \in \{0,1\}^*$,

$$\Pr[A(x) = L(x)] \geq 2/3$$

[2]Whether or not sufficiently many such uncorrelated and random bits can be found in nature is somewhat questionable; these questions are partially addressed in the study of randomized algorithms, specifically (but from different perspectives) in the areas of *derandomization* and *randomness extraction*.

where the probability is taken over the setting of the random tape of the algorithm A.

As we will see soon, the choice of the constant $2/3$ is irrelevant, as long as the constant is somewhat bigger than $1/2$. Notice that if we replace the constant $2/3$ with 1, we obtain exactly the class P. So in particular P $\subseteq$ BPP.

Certain randomized algorithms have a *one-sided error* property; they never accept "no" instances of the problem, but may fail on a small fraction of "yes" instances. For instance, if we ask the question "Is the minimum cut of G at most k?" and if the answer is "yes", Karger's algorithm will be correct for most choices of randomness, but if the answer is "no", it will be correct for all choices of the randomness.

Definition 2. The class RP consists of all decision problems L for which there exists a randomized algorithm A running in probabilistic polynomial-time and such that for every $x \in \{0,1\}^*$,

$$\begin{aligned} x \in L &\implies \Pr[A(x) = 1] \geq 1/2 \\ x \notin L &\implies \Pr[A(x) = 0] = 1. \end{aligned}$$

Again, the choice of constant $1/2$ is irrelevant, as long as this quantity is bounded away from zero. Similarly, we can have algorithms that are always correct on the "yes" instances but may err on some fraction on "no" instances. This is the class coRP. Observe that $L \in$ coRP iff $\overline{L} \in$ RP.

So far the algorithms we look at are always efficient and mostly correct. We can also consider algorithms that are mostly efficient, but always correct. Say that a randomized algorithm A runs in *expected polynomial time* if there is a polynomial p such that for all x, the expected running time of A on input x is at most $p(x)$.

Definition 3. The class ZPP consists of all decision problems L that are decided by algorithms A such that for all x, $\Pr[A(x) = L(x)] = 1$ and A runs in expected polynomial time.

Alternatively, ZPP can be defined as the class of decision problems L for which there exists a randomized algorithm B that always runs in polynomial time and on every input x outputs either $L(x)$ or the special symbol $'?'$, with the property that for every x,

$$\Pr[B(x) = L(x)] \geq 1/2.$$

Here is a sketch of the equivalence of the two definitions: To turn algorithm A into algorithm B, run $A(x)$ for at most $2p(|x|)$ steps; if A has finished within this time bound output the answer (which is guaranteed to be correct), otherwise output $'?'$. Conversely, given algorithm B, you can obtain A from B by running $B(x)$ sequentially for as many times as necessary until $B(x)$ outputs an answer different from $'?'$.

2 Amplifying the success probability

We now show that the constants that determine the success probability of randomized algorithms are indeed arbitrary. We focus on BPP algorithms. Suppose you have an algorithm for a decision problem L such that

$$\begin{aligned} x \in L &\implies \Pr[A(x) = 1] \geq c(|x|) \\ x \notin L &\implies \Pr[A(x) = 1] \leq s(|x|). \end{aligned}$$

If $c(n)$ and $s(n)$ are somewhat bounded away from one another, we can make c very close to 1 and s very close to zero by running the algorithm independently $m = \mathrm{poly}(n)$ times and observing what fraction of copies accept. If this number is closer to $m \cdot c(n)$ than to $m \cdot s(n)$ we accept, otherwise we reject.

Let X_i be an indicator random variable for the ith run of the algorithm accepting and $X = X_1 + \cdots + X_m$, the number of accepting trials. Let $\mu = \mathrm{E}[X]$. If $x \in L$, then $\mu \geq m \cdot c(n)$, while if $x \notin L$, then $\mu \leq m \cdot c(n)$.

Recall the Chernoff bound, which tells us that if m is large, then X is very close to its expectation with extremely high probability:

Theorem 4 (Chernoff bound)**.** *Let $X_1, X_2, \cdots, X_m$ are independent random variables with $\Pr[X_i = 1] = p, \Pr[X_i = 0] = 1 - p$. Set $X = X_1 + \cdots + X_m$ and $\mu = \mathrm{E}[X]$. Then*

$$\Pr\big[|X - pm| \geq \epsilon m\big] \leq 2e^{-\epsilon^2 m/2p}$$

We set $\epsilon = (c(n) - s(n))/2$ to obtain the following consequence:

$$\begin{aligned} x \in L &\implies \Pr[X \leq (c(n) + s(n))m/2] \leq e^{-\delta(n)m} \\ x \notin L &\implies \Pr[X \geq (c(n) + s(n))m/2] \leq e^{-\delta(n)m} \end{aligned}$$

where $\delta(n) = (c(n) - s(n))^2/2$. So as long as $c(n) - s(n) = n^{-\Omega(1)}$ – that is, $c(n)$ and $s(n)$ are not too close, we can set $m = p(n)/\delta(n)$, where p is an arbitrary polynomial. Then the algorithm that runs m independent copies of A and accepts whenever $(c(n) + s(n))m/2$ of those copies accepts is a polynomial-time algorithm that has failure probability at most $2^{-p(n)}$.

3 Relations between randomized complexity classes

While the different models of randomized computations BPP, RP, and ZPP were introduced for different reasons, they are closely related, and also related to P and NP. Here is how:

P $\subseteq$ ZPP: Any algorithm for a decision problem that runs in polynomial time in particular runs in expected polynomial time.

ZPP $\subseteq$ RP $\cap$ coRP: We first show ZPP $\subseteq$ RP. Consider the definition of ZPP where an algorithm B returns either the correct answer or $'?'$. When B outputs 0 or 1, output this answer; when B outputs $'?'$, output 0. So B is never wrong on "no" instances. The argument that ZPP $\subseteq$ coRP, is analogous: Now the algorithm always outputs 1 when in doubt.

$\mathrm{RP} \cap \mathrm{coRP} \subseteq \mathrm{ZPP}$: Given an RP algorithm A and a coRP algorithm B for the same problem, on input x, run both A and B. If $A(x)$ outputs 0, output 0; if $B(x)$ outputs 1, output 1; otherwise, output $'?'$. So this algortihm never outputs the wrong answer, and it only outputs $'?'$ with probability 1/4.

$\mathrm{RP} \subseteq \mathrm{BPP}$: By our discussion in the previous section, the probability 1/2 in the definition of RP can be replaced by 2/3.

$\mathrm{RP} \subseteq \mathrm{NP}$: Think of the randomness r used by the algorithm as an NP witness. If x is a "no" instance, then $A(x, r)$ rejects for all r. If x is a "yes" instance, then $A(x, r)$ accepts with nonzero probability, so there is some r for which $A(x, r)$ acccepts.

What is the relation between BPP and NP? If we believe that randomness is a realistic resource and NP problems are hard, NP should contain problems that are outside BPP. In fact, the notion of NP-completeness works for randomized algorithms as well, so the conjecture $\mathrm{NP} \not\subseteq \mathrm{BPP}$ is equivalent to saying $\mathrm{SAT} \notin \mathrm{BPP}$. But proving this could only be harder than proving $\mathrm{SAT} \notin \mathrm{P}$.

Let us now reverse the question: Is it possible that randomness can be used to decide some problems *outside* NP? The answer to this is not known, but it is believed that the answer is no. We will see some evidence for this later today, and also later in the course.

4 Adleman's Theorem

What is the non-uniform model of randomized efficient computation? Remember that deterministic (polynomial-time) Turing Machines were modeled by (polynomial-size) circuit families in the non-uniform setting. So maybe we should expect that randomized Turing Machines should be modeled by randomized circuit families. But in fact it turns out that randomness does not add any power to circuits. In particular we have the following simulation:

Theorem 5 (Adelman)**.** *$\mathrm{BPP} \subseteq \mathrm{P/poly}$.*

This result indicates that randomness is not too powerful. In particular, if we believe that SAT does not have polynomial-size circuits, then SAT cannot have polynomial-time randomized algorithms either.

Proof. Let $L \in \mathrm{BPP}$. By our discussion on amplifying the success probability, L has a BPP type algorithm such that for every x,

$$\Pr_r[A(x,r) = L(x)] \geq 1 - 1/2^{|x|+1}.$$

We fix the input size $|x| = n$ and show that the same string r_n can be good for all inputs of size n

simultaneously.

$$\begin{aligned}
\Pr_{r_n}[\forall x \in \{0,1\}^n, A(x,r_n) = L(x)] &= 1 - \Pr_{r_n}[\exists x \in \{0,1\}^n, A(x,r_n) \neq L(x)] \\
&\geq 1 - \sum_{x \in \{0,1\}^n} \Pr_{r_n}[A(x,r_n) \neq L(x)] \\
&\geq 1 - 2^n \cdot 2^{-n-1} = 1/2,
\end{aligned}$$

so there is a r_n of length poly(n) that behaves correctly on all length n inputs. Let circuit C_n simulate $A(x, r_n)$ on input x when $|x| = n$. Then C_n computes L, so $L \in$ P/poly. □

5 Randomness versus determinism

Perhaps the most interesting question about randomness is whether it can really help us speed up computations in a fundamental way. While Karger's algorithm did provide a more efficient alternative for finding minimum cuts in graphs, both of these algorithms run in polynomial time; so one can argue that in the grand scheme of things, these savings are not that important.

One way to ask the question is like this: How efficiently can we simulate randomized algorithms deterministically? The best deterministic simulation we know of randomized algorithms for decision problems comes from the trivial fact BPP $\subseteq$ EXP. Is a subexponential, or even polynomial-time simulation, of BPP-type algorithm possible? Let's review some evidence for this problem. There were some instances in the past where people came up with an efficient randomized algorithm for some problem, only to discover later that the same problem can be solved deterministically. One example is the problem problem of determining if a number is prime. This is a coNP question. In the 1970s it was first shown that this problem is in coRP, then in RP as well. Many years later – in 2002 – primality testing was discovered to be in P.

This story may be taken as evidence that if we work hard enough, we can remove randomness from algorithms that appear to require it. Other examples of this kind are algorithms for minimum-cut and finding perfect matchings in graphs (although in those cases, the deterministic algorithms were discovered first.)

To give contrasting evidence, here is an example of a problem that can be solved by a simple randomized polynomial-time algorithm, but for which the best known deterministic algorithm takes exponential time.

Polynomial Identity Testing (PIT)
Input: An arithmetic formula $F(x_1, ..., x_n)$ with integer coefficients
Problem: Is F identically zero?

An *arithmetic formula* over the integers is an expression built up from the variables $x_1, \ldots, x_n$, the integers, and the arithmetic operations $'+'$ and $'\times'$, for instance:

$$(x_1 + 4 \times x_3 \times x_4) \times \big((x_2 - x_3) \times (x_2 + x_3)\big) - 3 \times x_2.$$

We say that F is identically zero, in short $F \equiv 0$, if when we expand the formula and carry out all cancellations we obtain 0.

A deterministic algorithm that comes to mind for this problem is to work out the symbolic expansion of the polynomial and check if everything cancels out. This might take exponential time. A clever alternative is to carefully choose a set of points $(a_1, \ldots, a_n) \in \mathbb{Z}^n$ and evaluate $F(a_1, \ldots, a_n)$. If F does not evaluate to zero, then we can safely answer "no". But if F always evaluates to zero, it might either be that $F \equiv 0$, or maybe $F \not\equiv 0$ but our choice of evaluation points was unlucky and we always ended up with the zero polynomial. It is not known how to choose evaluation points deterministically in a manner that rules out the second possibility. However, it won't be hard to show that a *random* choice of points is likely to work.

Theorem 6. PIT $\notin$ coRP

Proof. Let m be the number of multiplications in F. Consider the following randomized algorithm for PIT:

```
A:  On input $F$,
    Choose values $a_1, \ldots, a_n$ independently at random from the set $\{1, \ldots, 2m\}$.
    Evaluate $b = F(a_1, \ldots, a_n)$ (over the integers).
    If $b \neq 0$, reject; otherwise, accept.
```

So, if $F \equiv 0$, this algorithm always accepts it. Now we show that if $F \neq 0$, then the algorithm rejects F with probability 1/2. We will need the following theorem.

Theorem 7 (Schwarz-Zippel)**.** *For any nonzero polynomial p of degree d over the integers and every set $S \subseteq \mathbb{Z}$,*

$$\Pr_{a_1,\ldots,a_n \sim S}[p(a_1, \ldots, a_n) = 0] \leq \frac{d}{|S|}$$

In our case, F is a polynomial of degree at most m and S is a set of size $2m$, so the algorithm will detect that $F \not\equiv 0$ with probability at least 1/2. □

Proof of Theorem 7. We prove it by induction on n. If $n = 1$, $Pr_{a \in S}[p(a) = 0] \leq \frac{d}{|S|}$, that is, there are at most d values $a_1, \ldots a_d$, which satisfy $p(a_i) = 0$.

Suppose not, then exist distinct $a_1, \ldots a_{d+1}$, which satisfies $p(a_1) = \ldots = p(a_{d+1}) = 0$, so $\prod_{i=1}^{d}(x - a_i)$ divides p and $\prod_{i=1}^{d+1}(x - a_i)$ also divides p, which conflicts.

If $n > 1$, we have

$$P(x_1, \ldots, x_n) = x_1^{d_1} \cdot p_{d_1}(x_2, \ldots, x_n) + x_1^{d_1 - 1} \cdot p_{d_1 - 1}(x_2, \ldots, x_n) + \ldots + p_0(x_2, \ldots, x_n)$$

So,

$$\Pr[p(a_1, \ldots, a_n) = 0] \leq \Pr[P_{d_1}(a_2, \ldots, a_n) = 0] + \Pr[p(a_1, \ldots, a_n) = 0 \mid p_{d_1}(a_2, \ldots, a_n) \neq 0]$$

By induction hypothesis and case $n = 1$,

$$\Pr[p_{d_1}(a_2, ..., a_n) = 0] \leq (d - d_1)/|S|$$
$$\Pr[p(a_1, \ldots, a_n) = 0 \mid p_{d_1}(a_2, ..., an) \neq 0] \leq d_1/|S|.$$

So,

$$\Pr[p(a_1, ..., a_n) = 0] \leq \frac{d}{|S|}. \qquad \square$$

Since we do not know of any subexponential time deterministic algorithm for polynomial identity testing, it may seem reasonable to conjecture that PIT *requires* exponential time. But then we know from work of Russell Impagliazzo and Avi Wigderson that BPP = EXP — namely *all* problems in exponential time, including SAT and many others even more difficult ones can be solved efficiently using randomness! This seems very unlikely. So although we don't *know* of any subexponential algorithm for polynomial identity testing, theory tells us that such an algorithm is quite likely to exist. (In fact, theory gives us a very good candidate for such an algorithm, but one whose correctness relies on reasonable yet unproven assumptions.)

What Impagliazzo and Wigderson actually proved is the following curious dichotomy: Either BPP = EXP, or every BPP algorithm can be simulated in subexponential time. We won't prove this result, but in the next lecture we will introduce one of the main ingredients in the proof – the Nisan-Wigderson pseudorandom generator.