Notes for CPSC 468/568: Computational Complexity

ZPP, Interactive Proofs and Arthur-Merlin games, and pairwise-independent hash functions

March 3, 2015

1 ZPP

We've already looked at BPP (bounded-error probabilistic polynomial time) – it has *two-sided error*; it allows an algorithm for language L to output 0 when $x \in L$ and vice-versa with some small probability (we use $\leq 1/3$, but it can be any value less than 1/2).

There's also RP, or randomized polynomial time, which has *one-sided error*. That is, $x \in L \rightarrow Pr[M(x) = 1] \geq 2/3$, but $x \notin L \rightarrow Pr[M(x) = 0] = 1$, where M runs in polynomial time. The full class is called RTIME (so $\cup_{c>0} RTIME(n^c) = RP$). RP basically always runs in polynomial time in the input size, always returns NO if the correct answer is NO, and returns YES with some probability (we use $\geq 2/3$), and sometimes (incorrectly; $\leq 1/3$) NO if the correct answer is YES. coRP has error in the opposite direction.

Note that $P \subseteq RP \subseteq NP$ (similarly, $P \subseteq coRP \subseteq coNP$), and that $RP \subseteq BPP$ (similarly, $coRP \subseteq BPP$), but we don't know if $BPP \subseteq NP$. It is not known whether these inclusions are strict. If P = BPP is true (which we probably believe), then RP, coRP, and P collapse. If we believe that $P \neq NP$, this then implies that RP is strictly contained in NP. We don't know whether RP = coRP, or whether RP is a subset of the intersection of NP and coNP (again, this would be implied by P = BPP).

Now that we've refreshed our memories on BPP and RP, let's talk about ZPP. $ZTIME(T(n))$ contains all languages L for which $\exists M$ that runs in expected time $O(T(n))$, and when it halts on input x, the output is $L(x)$. This is a Las Vegas algorithm (remember quicksort – you "gamble" with the resources, not with the output). Basically, it always returns the correct YES or NO answer and the running time is polynomial in expectation for every input. We can also define it as the class of languages for which we have a probabilistic machine M that always runs in polynomial time and returns an answer YES, NO or LLAMA (DO NOT KNOW). The answer is always either LLAMA with probability at most 1/2 or the correct answer.

$ZPP = RP \cap coRP$ – this is often taken to be the definition of ZPP. To show this, first note that every problem which is in both RP and coRP has a Las Vegas algorithm as follows: Suppose we have a language L recognized by both the RP algorithm A and the (possibly completely different) coRP algorithm B. Given an input in L, run A on the input for one step. If it returns YES, the answer must be YES. Otherwise, run B on the input for one step. If it returns NO, the answer must be NO. If neither occurs, repeat this step. Note that only one machine can ever give a wrong answer, and the chance of that machine giving the wrong answer during each repetition is at most 50%. This means that the chance of reaching the k^{th} round shrinks exponentially in k, so that the expected running time is polynomial. This shows that $RP \cap coRP$ is contained in ZPP. The proper proof is also very easy; we might give it to you for a HW assignment.

Other things: We don't know of any complete problems for BPP (end of page 137).

2 Interactive Proofs

What do we want from a "proof"? We want to be convinced if it is true, we want to be able to find fault with it if it is false, and, as I have written repeatedly in your homeworks, we want it to be **short**. A natural way to think of proofs is to have a prover P and a verifier V (they each have one obvious job). Now, it turns

out that thinking of problems in this way can lead to some very interesting results *as long as the verifier is randomized*. E.g., proving that a formula is not satisfiable is in coNP (in fact, it is coNP-complete), and so we don't think we have a traditional polynomial size proof for it. However, if we allow interaction, we can do so – not just for this problem, but for any problem in PSPACE.

This has very direct real-world uses – a common problem in cryptography is user-identification. A user wants to prove that it has a password without revealing said password to the server (the user may use the same password elsewhere, etc.). We want the server to essentially *interrogate* the user to convince itself that the user does indeed possess the correct password, and there are very interesting ways to do this. If fact, we have zero-knowledge proofs, which reveal *nothing* except for this one fact. Note that *interaction* means that P and V pass messages back and forth, ending with a message from P to V (there's no utility to having a final V to P message).

Recall the basic NP scenario – we're simply adding interaction to it. We need to ask ourselves whether we want to make P and V deterministic or probabilistic. If both are deterministic, things get pretty boring. For example, to verify membership in 3SAT, V asks P to announce the values of the literals in each clause, keeping track of the values to ensure that P doesn't produce contradictory values. The textbook provides some definitions about deterministic provers and verifiers; you should skim those.

Let's formalize the statement that we want to be convinced if a proof is true, we want to be able to find fault with it if it is false, and we want it to be **short** (and this is just NP): if $x \in L \rightarrow \exists P : out_V\langle V, P\rangle(x) = 1$, otherwise 0 (completeness: if $x \in L$, there is a proof that P can provide such that V accepts; soundness: $if x \notin L$, V always rejects the proof). What I really want to talk about is the computational power of P. We're going to let P be unbounded (the same way we treat adversaries in cryptography) – this makes sense because a false assertion should not be provable under any scenario. Since P is unbounded, determinism is a moot point (and we can see why a deterministic V would be even more boring than before).

Now, let's actually come to IP. We use a probabilistic V, *i.e.* V's questions will be based on some random coin-flips, and V is allowed to be wrong with some small probability. Why is this such a big deal? It turns out that this makes us jump from NP to PSPACE. The usual example used to illustrate this is as follows: Arthur, who is color blind, is getting ready for his coronation, when Merlin tells him that his socks don't match. Arthur is not entirely convinced; he's just met Bilbo and knows that wizards can be rather manipulative. To convince himself, he comes up with the following protocol (Arthur was a talented computer scientist): Merlin gives him the two (otherwise identical) socks, telling him, say, that the left one is red, and the other green and then turns around. Arthur tosses a coin, and switches the socks if it lands on tails. Arthur then asks Merlin to turn back and guess whether the socks have been switched. If Merlin can guess correctly on n repetitions, then Arthur will be convinced.

Let us now define this formally: For an integer $k \geq 1$ (that may depend upon the input length – ok, since P is unbounded), we say that a language L is in $IP[k]$ if $\exists$ a probabilistic polynomial time TM V that can have a k-round interaction with a function (think oracle!) $P : \{0,1\}^* \rightarrow \{0,1\}^*$ such that we have completeness ($x \in L \rightarrow \exists P : Pr[out_V\langle P, V\rangle(x) = 1] \geq 2/3$) and soundness ($x \notin L \rightarrow \forall P : Pr[out_V\langle P, V\rangle(x) = 1] \leq 1/3$). We define $IP = \cup_{c \geq 1} IP[n^c]$. All probabilities are over r, the coin-flips carried out by V. Note that if P were a PPT, we'd end up with BPP.

Example: Graph Non-isomorphism

This was what we thinly-disguised earlier as a color-blind Arthur with socks. Recall that two graphs are isomorphic if they are the identical up to a relabeling of the vertices. $GI \in NP$ (you can think why) – in fact, it is one of the most famous problems in NP. We do not know whether it is NP-complete (it turns out that if we could show that this is NP-complete, the PH would collapse). Here, we tackle the GNI problem – whether or not two graphs are non-isomorphic (you should think about where GNI fits).

We take two graphs G_0, G_1 and pick $i \in \{0,1\}$ u.a.r. We then randomly permute the vertices of G_i to produce a new graph H, which we send to P. P now has to guess which of the two graphs H is a permutation of – it is unbounded, so it can simply check all permutations if need be. P sends its guess j. We accept if $i = j$. If G_0, G_1 are isomorphic (the same color) then P cannot distinguish

between them. So, if $i = j$ always, then the graphs are not isomorphic; otherwise, P can do (at best) 1/2.

We can check to make sure that this fits the definition of IP. Note that if G_1 is not isomorphic to G_2, $\exists P : Pr[Vaccepts] = 1$; If not, P can do at best 1/2, which we can reduce to 1/3 by repetition.

It turns out that this definition is extremely robust. The idea was independently developed in the mid-1980s by Goldwasser, Micali, and Rackoff (who called it "interactive proofs"), and Babai and Moran (who called it "Arthur-Merlin Games"). Arthur is basically V, except for the fact that the coin tosses are public; Arthur goes first in AM games, and Merlin in MA games. You can think of $AM[k]$ as IP where the messages sent by A/V contain the transcripts of its coin tosses. Intuitively, we can see that having the coin tosses be public heavily restricts or reduces the kinds of computation we can do. Fortunately, this intuition is completely wrong – public/private coins are equivalent (the protocol might run for longer, but that's about it – Theorem 8.12; basically, Merlin can show Arthur that there are a large number of random strings on which to accept, reducing AM to IP). We can also restrict the number of rounds, have Arthur accept with Pr=1 (one-sided error), etc. GNI is actually in AM[2] (see Theorem 8.13).

We can also prove that $IP \subseteq PSPACE$ by seeing that P can maximize the probability of acceptance by V by computing recursively on all possible questions V can ask in each round (computing by induction on future rounds – remember, P is unbounded) in PSPACE (this is exercise 8.1).

3 Pairwise-independent Hash Functions

Recall the definition of pairwise independent random variables: If $\mathcal{S}$ is a finite set, and $X_1, \ldots, X_n$ are random variables assuming values in $\mathcal{S}$, then $X_1, \ldots, X_n$ are *pairwise independent* if for all $i \neq j$ and $a, b \in \mathcal{S}$, $Pr[X_i = a \wedge X_j = b] = Pr[X_i = a] \times Pr[X_j = b] = 1/|S|^2$.

Similarly, if we have two different but fixed strings x and x' of length n, we can choose a function h at random from $\mathcal{H}_{n,k}$, and end up with $\langle h(x), h(x')\rangle$ which is uniformly distributed over $\{0,1\}^k \times \{0,1\}^k$.

More formally, let $\mathcal{H}_{n,k}$ be a set of functions that map $\{0,1\}^n$ to $\{0,1\}^k$. We say that $\mathcal{H}_{n,k}$ is a *pairwise-independent hash-function family* if, for all $x \neq x'$ in $\{0,1\}^n$ and all y and y' in $\{0,1\}^k$,

$$\text{Prob}_{h \in_R \mathcal{H}_{n,k}} \left(h(x) = y \text{ and } h(x') = y'\right) = \frac{1}{2^{2k}}.$$

Equivalently, for any pair of distinct elements x and x' in $\{0,1\}^n$, if an element h is chosen uniformly at random from $\mathcal{H}_{n,k}$, the induced random variable $(h(x), h(x'))$ is uniformly distributed on $\{0,1\}^k \times \{0,1\}^k$.

Obviously, the set of *all* functions from $\{0,1\}^n$ to $\{0,1\}^k$ is a pairwise-independent hash-function family. In order to be useful, however, elements of $\mathcal{H}_{n,k}$ should have polynomial-length representations (so that one can choose one uniformly at random by flipping a polynomial number of coins) and should be computable in polynomial time. We now specify one such family. There are others with these two desirable properties.

Recall that elements of the finite field GF(2^n) can be represented by n-bit strings. It is easy to see that the set of all $h_{a,b}$, where a and b are both elements of GF(2^n) and $h_{a,b}(z) = az + b$, is a pairwise-independent hash-function family $\mathcal{H}_{n,n}$. First, note that, as a and b range over all of GF(2^n), the function $h_{a,b}$ ranges over all affine functions that map GF(2^n) to GF(2^n), of which there are 2^{2n}. Choosing a and b independently and uniformly at random is tantamount to choosing such an affine function uniformly at random. On the other hand, each quadruple x, x', y, y' of elements of GF(2^n) such that $x \neq x'$ uniquely determines an affine function h such that $h(x) = y$ and $h(x') = y'$. (Just let $h(z) = (\frac{y'-y}{x'-x})z + (y - (\frac{y'-y}{x'-x})x)$.) For a given x, x', y, y' such that $x \neq x'$, the probability that an $h_{a,b}$ chosen uniformly at random is equal to this h is exactly 2^{-2n}, which is what we need for $\mathcal{H}_{n,n}$ to be a pairwise-independent hash-function family.

If $k > n$, we can get $\mathcal{H}_{n,k}$ by using $\mathcal{H}_{k,k}$ and padding the input strings with $n - k$ zeroes. If $k < n$, we can get $\mathcal{H}_{n,k}$ by using $\mathcal{H}_{n,n}$ and chopping off the last $n - k$ output bits.