6.841/18.405J: Advanced Complexity Theory	May 1, 2002
Lecture 20	
Lecturer: Madhu Sudan	*Scribe: Eleni Drinea*

Today we will first elaborate on the definition of Distributed NP, introduced in the last lecture. We will then define what it means for a problem to be in Avg-P and start discussing a completeness result for DNP.

1 Definitional Issues

In the definition of a distributional problem in the last lecture, the input distribution was a single distribution on all inputs of all sizes. Equivalently, according to Impagliazzo, we can think of the input distribution as being on a finite set of possible inputs (e.g., of at most some fixed size n). Thus for today's lecture, the distribution D we will work with is $D = \{D_n\}_{m=1}^{\infty}$, i.e., a collection of distributions D_n.

There are two classes of distributions that we are interested in:

- P-computable distributions D: let $\tilde{D} = \sum_{y \leq x} D(y)$, where we think of the ordering "$y \leq x$" on $\{0,1\}^n$. We say that D is P-computable if $\tilde{D}$ is polynomial time computable. This is the nicest class of distributions one can think of. Recall that we defined a pair (L, D) to be a decision problem in DNP if $L \in NP$ and D is P-computable.
- P-samplable distributions D: this is a quite elaborate class of distributions. We say that D is P-samplable if there exists a polynomial time algorithm A that outputs x with probability $D(x)$. Note that a P-samplable distribution need not be P-computable. For example, if we pick an assigment $x \in \{0,1\}^m$ on m variables, and construct a formula F on (say) $4m$ clauses that is satisfied by x, then the distribution on such formulae is P-samplable. However, in order to compute $\tilde{D}(F)$, we probably need #P power in order to compute the probability of each $F' < F$ (since we need to count the number of assignments that satisfy F'). Thus this distribution is probably not P-computable.

2 δ-good algorithms and Avg-P

We define problems in *Distributed NP* to be pairs (R, D) such that R is a polynomial time computable binary relation $R(x, y)$ and D is a distribution on x (the first of R's arguments). Given x distributed according to D, we want to find a y such that $R(x, y)$, if such a y exists. A notion that is related to how well we want to solve this search problem is Avg-P.

A problem is in Avg-P if there exists an "efficient" algorithm B that "solves" this problem. Of course, we need to be more specific about the notions in quotes above. For the notion of "solvability", δ-good algorithms are satisfactory.

Definition 1 *An algorithm A is δ-good for R and D if*

$$\Pr_{x \leftarrow D}\left[\begin{array}{l} \textit{if } \exists y \textit{ s.t. } R(x,y), \textit{ then } R(x, A(x)) \textit{ is true} \\ \textit{and} \\ \textit{if } \forall y\ \neg R(x,y), \textit{ then } A(x) = \textit{"error"} \end{array}\right] \geq 1 - \delta$$

Note that we are using only benign algorithms, that never make errors. However, they may need more time to produce a definite answer; in this case they output "?". In other words, A given x can produce three outputs: y, in which case $R(x, y)$ is true, "error", in which case there is no y such that $R(x, y)$, or "?", if A does not have enough time to decide.

Now let's look at the efficiency requirements for algorithms for problems in Avg-P. For example, consider an algorithm A that solves a problem (R, D) in the following way: with probability $1 - \frac{1}{2\sqrt{n}}$ it takes time n and with probability $\frac{1}{2\sqrt{n}}$ it takes time $t = 2^n$. The expected running time of the algorithm is $2^{\Theta(n)}$; then A

is a *bad* algorithm for the problem if our criterion is the expected running time. However if we change t to $2^{n^{1/3}}$, then (under the same criterion) A becomes *good*. Similarly, if A runs on a 2-tape TM and $t = 2^{.75\sqrt{m}}$ time, then A is *good*. However if we simulate this algorithm on an 1-tape TM, with probability $\frac{1}{2^{\sqrt{n}}}$ we will need time $2^{1.5\sqrt{m}}$ (because of the quadratic overhead of the simulation). Thus the expected running time of the simulation is exponential in n and the algorithm will now be considered *bad*.

The observations so far already suggest that the expected running time is not suitable as a criterion for the efficiency of an algorithm that solves a problem in Avg-P. First it is hardwired in the model of computation; and even polynomial changes in the running time do not maintain the property of *goodness*. Moreover, using such algorithms may cause problems in the composition of reductions. In particular, suppose $(R, D) \in$ Avg-P. Then there may exist (R', D') that reduces in polynomial time to (R, D) and yet, $(R', D') \notin$ Avg-P. This is certainly something that the notion of reduction should not allow.

The above discussion leads us to the following definition for Avg-P:

Definition 2 *A problem (R, D) is in Avg-P if there exists an algorithm B on two inputs, x and δ, such that $B(\cdot, \delta)$ is δ-good for (R, D) and B runs in time polynomial in the length of x and in $1/\delta$.*

This definition of Avg-P is robust (e.g., the problem with the reductions no longer exists) and also makes sense, as the running time increases when we increase the probability that the algorithm returns a definite answer (i.e., we decrease δ).

3 Towards a completeness result

The main question that arises at this point is whether DNP $\subseteq$ Avg-P, with the distributions considered either P-computable or P-samplable. The reasonable way to go about this issue is to define a complete problem in DNP and ask whether this is in Avg-P.

3.1 α-dominance between distributions

Before we actually address this question, it is reasonable to ask whether DNP problems with P-samplable distributions are harder than DNP problems with P-computable distributions. Impagliazzo and Levin proved that this is not the case; every DNP problem complete for P-computable distributions is also complete for all samplable distributions. In particular, starting with problem (R, D), where R is an arbitrary (polynomial time) relation and D is P-samplable, we can reduce this to some problem $(R'_{(R,D)}, D')$, where D' is P-computable, e.g. approximately uniform. (We will soon discuss what is means to reduce a distributional problem to another distributional problem.)

Before the Levin-Impagliazzo theorem, Levin proved that every problem (R, D), where D is P-computable, can be reduced to (Π, U), where Π is a fixed problem and U is a uniform distribution. Thus there is a complete problem for DNP, with P-computable distributions.

Our goal for now is to give the high-level description of how the reduction from DNP with P-samplable distributions to DNP with P-computable distributions works. However, before we get to the actual reduction, we will discuss what it means to "reduce" a distributional problem to another.

From now on, we will be thinking of D as being a sampling algorithm. I.e., D gets x, which is a uniformly distributed n-lettered string and outputs an n-lettered string y distributed according to D.

Now suppose we are given an instance of (R, D) and want to reduce it to an instance of (R', D'). This may be too strong to require in general and we do not really need to achieve that much: maybe we are satisfied if the instances of (R', D') are not produced exactly according to D' but rather according to some D'' which is nicely related to D'. In other words, we would like to say that if A is an algorithm that is δ-good for (R', D') then A is also δ'-good for some (R', D''), given that D'' is related in a certain way to D'. It turns out the right way to formulate this relation is the notion of α-dominance.

Definition 3 *We say that a distribution D_1 α-dominates a distribution D_2 if for all x*

$$D_1(x) \geq \frac{D_2(x)}{\alpha}$$

The intuition here is that if A is good on some D_1, then it will still be good on some D_2 that is α-dominated by D_1. This is formally stated in the following theorem.

Theorem 4 *If A is δ-good for (R, D_1) and D_1 α-dominates D_2, then A is $\alpha\delta$-good for (R, D_2).*

Recall that in Avg-P we are in control of the δ; for example, if D_1 has a bad α dominance over D_2, then we can adjust δ so that $\alpha\delta$ is such that the algorithm is still good enough for (R, D_2).

3.2 The Impagliazzo-Levin Reduction

Let us consider the distribution D_m that picks an integer k uniformly at random from the set $\{1, \ldots, n\}$ and outputs (k, w), where w is chosen uniformly at random from $\{0,1\}^k$. This distribution is essentially "uniform". In general, consider D_n that outputs a collection of tuples, such that the first element of the tuple specifies the rest. For example, $D_n : (k, x, y, z, i, w)$, where x, y, z have length k, i is an integer in $1, \ldots, k$ and w has length i. Then these are certainly P-computable distributions; we will also consider these as uniform distributions.

Our goal is to reduce a problem (R, D), supposedly hard, where R is arbitrary and D is P-samplable, to a problem (R', D'), where D' is uniform (as previously discussed). In this setup, the universe picks $z \in \{0,1\}^n$, applies D and outputs $D(z) = x \in \{0,1\}^n$. Now we need to find R' such that (R', D') is hard, with D' being a uniform distribution.

A first attempt for R' would be $R'(z, y) = R(D(z), y)$. Clearly R' is polynomial time computable since D is P-samplable and if R is hard on D, so is R' on the uniform distribution. However, this attempt fails, in that, looking at the formal reduction, given x, we should find some z such that $D(z) = x$. But in general $D^{-1}(x)$ may not be tractable (e.g., as we mentined earlier today, z could be an assignment over m variables and $D(z) = x$ a formula on a certain number of clauses satisfied by z).

The idea that works is to implicitly (rather than explicitly) specify z. This means the following: suppose all the preimages of x are in the set S, which consists of 2^k elements; suppose further that z is the i-th element in S. Then z can be specified by (x, k, i), where $i \in \{0,1\}^k$ is the index of z in S.

However, enumerating all the elements in S is tricky. What we do instead is the following: we define the distribution D_2 that outputs tuples of the form

$$(x, h, k, w),$$

where a z is picked from the n-lettered universe and then $x = D(z)$ is computed; h is a randomly generated hash function on the n-lettered strings; k is uniformly chosen from $\{0, \ldots, n\}$ and is a guess for the logarithm of the number of preimages of x; $h(z) = w \in \{0,1\}^k$ (we can think of applying h on z and then just look at the first k coordinates). We claim that if solving R on D is hard, then solving R on D_2 is also hard.

Now we define a uniform distribution D_1 that outputs tuples

$$(x, h, k, w),$$

where x is uniformly chosen from the n lettered strings, h is randomly generated, $k \in \{1, \ldots, n\}$, and w is uniformly chosen from $\{0,1\}^k$. We also define R' as

$$R'\left((x, h, k, w), (y, z)\right) = R(x, y) \text{AND} \left(h(z) = w, D(z) = x\right)$$

where $h(z)$ is restricted to the first k coordinates.

The interesting thing about defining D_2 is that now we can claim that R' on D_1 is as hard to solve as R on D_2. The basic idea behind the proof is that D_1 dominates D_2 within a polynomial factor.