6.841/18.405J: Advanced Complexity Theory	April 29, 2002
Lecture 19	
Lecturer: Madhu Sudan	*Scribe: Christos Kapoutsis*

1 Introduction

In the complexity-theoretic terms we have seen so far, the formal way to express the intuitive fact that "a problem is very difficult to practically solve by a computer" is to say that "the problem is NP-hard" or, if we restrict our attention to just problems in NP, to say that "the problem is NP-complete". If we indeed accept that this matching between intuition and formalism is a correct one and we further assume that P≠NP, then it seems like we should also accept the following two conclusions:

- If a formal problem correctly describes a practical one and also proves to be NP-complete, then we should stop looking for efficient solutions to the practical problem.
- Cryptography is feasible.

The truth is that neither conclusion can be derived without additional assumptions. Today we discuss why this is so for the first of them.

The motivation comes from real-world examples of algorithms that solve NP-complete problems in a "satisfactory" way, in all practical respects. By studying what "satisfactory" means in those cases, we see that two important choices that we made when we defined *what it means for an algorithm to solve a problem* are actually not necessary, as far as practical solvability is concerned:

- **Exact solution:** We required that for every instance of the problem the algorithm should return the *exact* answer. In certain practical cases, we are just as happy even with some *approximation* of this exact answer.
- **Worst case instances:** We required that the algorithm should return the exact answer for *every* instance of the problem. In certain practical cases, hard instances (i.e., the instances responsible for the problem being NP-complete) rarely happen, and we are just as happy even with an exact answer *most of the times*.

In the next two sections we see what modifications in our theory so far are appropriate for the study of the above cases.

2 Approximation algorithms

Formally, a *combinatorial optimization problem* is any polynomial-time computable function $f : \{0,1\}^* \times \{0,1\}^* \to \mathbb{Z}^+$ mapping every instance $x \in \{0,1\}^*$ of the problem and every possible answer $y \in \{0,1\}^*$ to it to some value $f(x,y) \in \mathbb{Z}^+$ indicating how "good" y is as a solution to x. An algorithm A *solves* f if, for every x, $A(x)$ is $f(x,y)$ for the best solution y to x,

$$A(x) = \max_y \{f(x,y)\}.$$

Sometimes $f(x,y)$ indicates how "bad" y is as a solution to x, in which case it is the minimun we are after, instead of the maximum.

For example, the problem of finding the size of the maximum clique in a graph is defined by the function MAXCLIQUE mapping every graph $G = (V,E)$ and every $c \subseteq V$ to the value

$$\text{MAXCLIQUE}(G,c) = \begin{cases} 0 & \text{if } c \text{ is not a clique in } G, \\ \text{the size of } c & \text{otherwise,} \end{cases}$$

and we want to maximize $\text{MAXCLIQUE}(G,\cdot)$. Similarly, the optimization problem of finding the chromatic number of a graph is defined by the function CHROMATIC mapping every graph $G=(V,E)$ and every coloring $c: V \to \mathbb{N}$ of its vertices to the value

$$\text{CHROMATIC}(G,c) = \begin{cases} \infty & \text{if } c \text{ is an illegal coloring,} \\ \text{the number of colors used by } c & \text{otherwise,} \end{cases}$$

and we want to minimize $\text{CHROMATIC}(G,\cdot)$. The problems MAXSAT and MAX3SAT are also defined in a similar manner.

Relaxing the requirement that the exact optimum is returned, we get the notion of an *α-approximating algorithm for a problem f*, where α describes the proximity to the exact optimum that we want to achieve as a function from input sizes to reals not less than 1. We say A is such an algorithm if, for every instance x of the problem,

$$\frac{\max_y\{f(x,y)\}}{\alpha(|x|)} \leq A(x) \leq \max_y\{f(x,y)\}.$$

In general, we say that any pair (f,α) defines an *approximability problem* and that A *solves* (f,α) if A α-approximates f.

Clearly, the step from optimization problems to approximability problems can be seen as the step from considering only the fixed approximation ratio $\alpha(n)=1$ to allowing other approximation ratios as well. Work by Johnson in the late 70's showed that this view offers little help: Problems that behave similarly for $\alpha(n)=1$ may exhibit radically different behaviour for other approximation ratios. For example, for f any of the problems MAXSAT, MAXCLIQUE, CHROMATIC, we know that the approximability problem $(f,1)$ is NP-complete. However:

- $(\text{MAXSAT}, 2)$ is in P.
- For all $\epsilon > 0$ and $c > 1$, $(\text{MAXCLIQUE}, 1+\epsilon)$ is as hard as $(\text{MAXCLIQUE}, c)$. That is, if one can approximate the maximum clique size of a graph within a constant factor, one can also approximate it with a ratio as close to 1 as desired.
- $(\text{CHROMATIC}, \frac{6}{5})$ is NP-complete.

This clearly shows that reasoning like $(f,\alpha) \equiv (g,\alpha) \implies (f,\alpha') \equiv (g,\alpha')$ is wrong.

The PCP methodology has provided techniques for studying approximation problems, and some of the results known today are:

- $(\text{MAXSAT}, \frac{8}{7})$ is NP-complete.
- $(\text{MAX3SAT}, \frac{8}{7}-\epsilon)$ is NP-complete, for all $\epsilon > 0$.
- $(\text{MAX3SAT}, \frac{8}{7}+\epsilon)$ is in P, for all $\epsilon > 0$.
- $(\text{MAXCLIQUE}, n^{1-\epsilon})$ is NP-complete, for all $\epsilon > 0$.
- $(\text{CHROMATIC}, n^{1-\epsilon})$ is NP-complete, for all $\epsilon > 0$.

3 Average case complexity

Formally, a *distributional decision problem* is a pair (L,D), where $L \subseteq \{0,1\}^*$ is a language and D is a distribution over $\{0,1\}^*$. (That is, D is a function $D: \{0,1\}^* \to [0,1]$ such that $\sum_{x\in\{0,1\}^*} D(x) = 1$.)

To solve the problem, an algorithm has to correctly determine membership in L when the inputs are drawn according to D. The algorithm is considered efficient if it "runs in expected polynomial time". The quotes indicate that the notion of efficiency is tricky to formalize, especially when one considers problems in NP. Work on how to do this correctly was done by Levin (1984).

3.1 Random self-reducibility of the permanent

For problems of high complexity (PSPACE, #P), the notions seem to be clearer. We begin the study of average case complexity with a theorem that computing the permanent is as hard on average as in worst case. Recall that a permanent of a matrix $M = [m_{i,j}]$ is given by

$$\text{perm}(M) = \sum_{\pi \in S_n} \prod_{i=1}^{n} m_{i,\pi(i)},$$

where S_n is the set of all permutations on $[n] = \{0, 1, \ldots, n-1\}$.

We start with the definition of a distribution D over matrices. For every $n \in \mathbb{Z}^+$, consider the following random experiment:

1. randomly select a prime p of size[1] $\Theta(n^{10})$, then
2. randomly select a matrix from $\mathbb{Z}_p^{n \times n}$ (i.e., construct an $n \times n$ matrix by randomly selecting each one of its elements from $\mathbb{Z}_p$).

Let D_n be the distribution over matrices defined by this experiment; then let $D = (D_n)_{n \geq 1}$. [Note that this doesn't exactly match the definition of a distribution over $\{0,1\}^*$ given above. This is ok, as it won't harm the correctness of the theorem. Alternatively, instead of one separate experiment for every $n \in \mathbb{Z}^+$, we could have only one experiment:

0′. randomly select n from $\mathbb{Z}^+$ with probability $\frac{1}{n(n+1)}$, then

1′. do Step 1 from above for the selected n, then

2′. do Step 2 from above for the selected n,

and have D be the distribution over matrices defined by this experiment (this is the correct definition according to Levin).]

The theorem we are going to prove is the following:

Theorem 1 If *there exists a polynomial-time algorithm A such that, for all sufficiently large n and* for matrices drawn from D_n, *A computes the permanent correctly with probability* $1 - \frac{1}{\text{poly}(n)}$, then *there is also a randomized, polynomial-time algorithm B that, for all sufficiently large n and* for every matrix M in D_n, *B computes the permanent of M correctly with high probability.*

Proof We don't get into the issues concerning the random selection of the prime number p. We consider a fixed p of size $\Theta(n^{10})$ and assume that a polynomial-time algorithm $A(\cdot, p)$ computes the permanent correctly with probability $1 - \frac{1}{\text{poly}(n)}$ for matrices randomly selected from $Z_p^{n \times n}$. Consider the algorithm $B(\cdot, p)$ that on input $M \in \mathbb{Z}_p^{n \times n}$ computes as follows:

1. picks a random matrix R from $\mathbb{Z}_p^{n \times n}$,
2. for each $i = 1, 2, \ldots, n+1$, computes $y_i \leftarrow A(M + i \cdot R, p)$ (where $i \cdot R$ stands for multiplying every entry of R by i),
3. interpolates and finds a polynomial $q(x)$ of degree n such that $q(i) = y_i$, for all $i = 1, 2, \ldots, n+1$.
4. outputs $q(0)$.

It can be shown that with high probability over its internal coin tosses, $B(\cdot, p)$ correctly computes the permanent of its input. ■

Note that the proof uses the idea of "self-correction" that we have seen in previous lectures: symbolically, perm(M) is a degree n polynomial in the n^2 variables $m_{1,1}, m_{1,2}, \ldots, m_{n,n}$; and A is an oracle that computes this polynomial correctly quite often; so, B is what we get by applying self-correction to A.

[1] "Size" here is the actual size of the number; not its length in bits.

3.2 The class DNP

In this section we give the definition of the class DNP (Distributional NP). In the next lectures we will give theorems about it.

First, we need to associate with every distribution D the function $\tilde{D} : \{0,1\}^* \to [0,1]$, defined by

$$\tilde{D}(x) = \sum_{y \leq x} D(y),$$

so that $\tilde{D}(x)$ is the probability that a string selected according to D "comes before" x. Here, a fixed, reasonable liner ordering of $\{0,1\}^*$ is assumed; e.g., the usual ordering $0, 1, 00, 01, 10, 11, \ldots$.

Now, we are interested in two special kinds of distributions over $\{0,1\}^*$.

Definition 2 *A distribution D over $\{0,1\}^*$ is P-computable if $\tilde{D}$ is polynomial-time computable (i.e., there exists a polynomial-time algorithm that on input x outputs $\tilde{D}(x)$).*

An example of such a distribution is the *uniform* distribution, defined by $D(x) = \frac{1}{|x|(|x|+1)} 2^{-|x|}$.

Definition 3 *A distribution D over $\{0,1\}^*$ is P-samplable if there exists a probabilistic algorithm A and a polynomial p such that A outputs x with probability $D(x)$ and after $p(|x|)$ steps.*

A P-computable distribution is P-samplable, as well. But the converse is probably false.

Definition 4 *The class DNP contains exactly the distributional decision problems (L, D) for which the language L is in NP and the distribution D is P-computable.*

Note that considering NP languages with P-samplable distributions results in a class larger than DNP.