29. Application of list decoding - Hardness of the Permanent

29.1 Introduction

In this lecture we will study the permanent function and its computational complexity. We will use ideas from the decoding and list decoding of Reed Solomon codes to get results about the average case complexity of the permanent function.

29.2 The permanent function PERM_n

Let A be an $n \times n$ matrix, say over the integers. The function $\mathsf{PERM}_n(A)$ is defined as follows:

$$\mathsf{PERM}_n(A) = \sum_{\sigma \in S_n} \prod_{i \in [n]} A_{i,\sigma(i)}.$$

The permanent function is a close cousin of the determinant function which is defined as follows:

$$\mathsf{DET}_n(A) = \sum_{\sigma \in S_n} (-1)^{\mathrm{sgn}(\sigma)} \prod_{i \in [n]} A_{i,\sigma(i)}.$$

While the definitions are almost identical, they are quite different in terms of the time complexity of computing the function. The determinant of a matrix is computable in poly time, whereas we don't know of any polynomial-time algorithms to compute the permanent. Valiant proved that the permanent is #P-hard, even in the case when A is a matrix with $0-1$ entries. This means that computing the permanent of a $0-1$ matrix is at least as hard as counting the number of satisfying assignments of a boolean formula. The class of problems in #P are at least as hard as NP. In fact, Toda proved that every problem in the polynomial hierarchy PH can be solved by a polynomial-time algorithm with oracle access to #P. In other words, every problem in PH can be solved by a polynomial-time algorithm that is allowed to make calls to a algorithm that computes the permanent function.

Notice that when we think of the matrix A as a bipartite graph, then the permanent function calculates the number of perfect matchings in the bipartite graph. While there is a polynomial-time algorithm to check if a bipartite graph has a perfect matching, counting the number of perfect matchings is as hard computing the permanent of a $0-1$ matrix.

Self-reducibility

Computing the permanent of an $n \times n$ matrix can be reduced to computing the permanent of smaller matrices in a very natural way. It follows from the definition of the function. In fact, we can write $\mathsf{PERM}_n(A) = A_{1,1}\mathsf{PERM}_{n-1}(A^{1,1}) + A_{1,2}\mathsf{PERM}_{n-1}(A^{1,2}) + \cdots +$

$A_{1,n}\mathsf{PERM}_{n-1}(A^{1,n})$ where $A^{1,j}$ is the $(n-1)\times(n-1)$ matrix with the the 1^{st} row and j^{th} column removed.

We will now see that the permanent function also has the *random self-reducibility* property as well and this will be used to show that computing the permanent is computationally hard even in the *average case*.

29.3 Permanent - average case to worst case

First, we will prove a result of Lipton that shows that if there is an algorithm that computes the permanent function for most matrices, then there is an algorithm that, with high probability, computes permanent for every matrix. Let $M_{n,p}$ denote the set of $n \times n$ matrices with entries from $\mathbb{F}_p$ for a large enough prime p. We will prove the following theorem.

Theorem 29.1 (Lipton '91)**.** *Let $\mathcal{A}$ be an algorithm with the following property:*

$$\Pr_{B\in_r M_{n,p}}[\mathcal{A}(B) = \mathsf{PERM}_n(B)] \geq 1 - \frac{1}{3(n+1)}.$$

Then, there is a randomized algorithm $\mathcal{A}'$ with the following property: For every $n \times n$ matrix C, we have

$$\Pr[\mathcal{A}'(C) = \mathsf{PERM}_n(C)] \geq \frac{2}{3},$$

where the probability is over the randomness of the algorithm $\mathcal{A}'$.