CS6840: Advanced Complexity Theory Jan 13, 2012

Lecture 6 : #P-Completeness

Lecturer: Jayalal Sarma M.N. *Scribe: Sunil K S.*

Thus we motivated to answer the questions. We saw $(\#\mathsf{P} = \mathsf{FP}) \iff (\mathsf{P} = \mathsf{PP})$ in decision world. This motivates understanding structure in the counting world. We start with the notion of reductions.

1 Reduction in Counting World

Our aim is to come up with a notion of #P-completeness and show structural classification of counting problems. Informally, we have seen counting problems that are hard to solve. We want our notion of hardness to capture this as closely as possible. We consider some natural options for such a definition and proceed from there.

Attempt 1: Let A,B be two languages. In decision world $A \leq_m^p B$ if and only if

1. $x \in A \iff \sigma \in B$.
2. $\sigma : \Sigma^* \to \Sigma^*$ is polynomial time computable.

Translating the above statements into counting scenario.

1. σ is polynomial time computable.
2. $f(x) = g(\sigma(x))$.

Two lectures back we showed SAT can be decided if $\#CYCLE$ can be computed. But following the above attempted notion of a reduction, we get the following.

$$\#\mathsf{SAT}(\phi) = \#CYCLE(\sigma(\phi)) \tag{1}$$

But if such a reduction exists, then we can solve SAT by checking if $\#SAT(\phi) = \#CYCLE(\sigma(\phi))$, which in turn can be done in polynomial time. Thus, if we impose this strictness then in #P only counting versions of NP-complete problems will be hard. Notice that the issue here is that the reduction is set to preserve the number of certificates !.

Let us make some preliminary observations. What if we allow factors in equation1? Could this save us? No, still if such a reduction exists, we can decide SAT by counting the value of $\#CYCLE$. Indeed, there would not have been a problem if there was a "+1" on the right hand side of the above expression, such that the zeroness of #SAT

does not carry over to the zeroness of $\#CYCLE(\sigma(\phi))$. This shows that may be we should allow non-trivial computations after the query to the function $\#CYCLE$. Since we would also like structural composibility and transitivity of reductions, this naturally leads to the attempt 2, in its generality.

Attempt 2: We attempt now a generalization of the notion of Turing reductions.

Definition 1. We say that a function $f \leq g$ if $f \in FP^g$. That is, if there exists a functional oracle Turing machine[1] with queries to $g : \Sigma^* \to \mathbb{N}$ that given any $x \in \Sigma^*$ can compute $f(x)$.
A function g is #P-hard if $\forall f \in \#\mathsf{P}, f \leq_m^p g$. g is complete for #P if $g \in \#\mathsf{P}$ and g is #P-hard.

Theorem 2. #SAT *is* #P*-complete.*

1. #SAT $\in$ #P: $\exists$ a non-deterministic turing machine polynomial time which guesses assignments (bit by bit), verifies the same, accepts if it satisfies the formula, and rejects otherwise. No assignment is guessed by two different non-deterministic computation paths. Hence the number of accepting paths will precisely be equal.

2. $\forall f \in \#\mathsf{P}, f \leq_m^p \#\mathsf{SAT}$: Let $f \in \#\mathsf{P}$ via a machine M such that $\forall x \in \Sigma^*, f(x) = \#acc_M(x)$.

   Cook-Levin Theorem - Revisited.

   $$L \in \mathsf{NP} \implies L \leq_m^p \mathsf{SAT}.$$

   Let $L \in \mathsf{NP}$, via machine M such that $x \in L \implies M$ has an accepting path. We construct $\phi_{(M,x)}$ from computation history such that M has an accepting path implies $\phi_{(M,x)}$ is satisfiable. For every accepting path, we have a satisfying assignment and for every satisfying assignment there is a corresponding accepting path too. Moreover, this map is There is a one-to one mapping between the set of accepting paths and the set of satisfying assignments. The certificates corresponds to the first set is the choice of non-deterministic bits and for the second set is the satisfying assignments. Such reductions are called *Parsimonious reductions*. An additional point is that this certificate bijection is polynomial time computable. That is given an accepting path of the machine M the reduction also gives you a way to transform it into a satisfying assignment of the formula and vice versa. Reductions satisfying the latter property are called *Levin reductions*. Note that, in order to prove NP-completeness, the reduction neither need to be parsimonious nor it should be Levin reduction. It is an additional structural property that the reductions seems to satisfy. Interestingly, the NP-complete

[1] A functional oracle TM is similar to the normal oracle Turing machine, but since the output of the oracle query is not a 1-bit, instead of q_{yes}/q_{no} as the two states to which the machine moves after the oracle answers, the machine has has an oracle output tape for the oracle to write the function value.

reductions that we have seen in the last course (like the SAT to 3SAT, 3SAT to independent set, Independent set to Vertex Cover), has this additional structural property. Thus all of them are #P-complete by our definition.

We have already talked out one example of a decision problem which is in P, but the counting version seems to be as hard as NP. We will consider another similar problem and show that the counting version can be shown to be #P-complete as well. Indeed $\#CYCLE$ is also #P-complete. We introduce the problem now:

2 Perfect Matching in a graph

Definition 3. : Perfect Matching: Let $G = (X, Y, E)$ be a bipartite graph. A subset $S \subseteq E$ is said to be a perfect matching iff

$$\forall u \in X \cup Y : \exists! v : (u, v) \in S$$

In words, each vertex has exactly one edge from S incident on it.

We will show a connection that the perfect matching has, with the following combinatorial parameter of the bipartite adjacency matrix. We state this parameter for an arbitrary matrix first.

Definition 4. : Permanent of a Matrix For an $n \times n$ matrix A,

Determinant of A, $Det(A) = \sum_{\sigma \text{ is a permutation} \in S_n} (-1)^{sign(\sigma)} \prod_{i=1}^{n} A_{i,\sigma(i)}$

Permanent of A, $Per(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} A_{i,\sigma(i)}$

Notice that the parameter is very similar to the determinant of the matrix A which is written as $Det(A) = \sum_{\sigma \in S_n} (-1)^{sgn(\sigma)} \prod_{i=1}^{n} A_{i,\sigma(i)}$ where $sgn(\sigma)$ assigns a sign to each term based on the sign of the permutation. Observing the differences, permanent of the 0-1 matrix can never be negative, where as that of a determinant of a 0-1 matrix can be negative. Determinant can be computed in FP, whereas we will show the permanent cannot be computed efficiently unless #P = FP.

For a bipartite graph G, let A be the bipartite adjacency matrix in which rows are indexed by x and columns are indexed by y. Note that here $|x| = |y|$, otherwise graph does not have a perfect matching. Now we will show the following connection:

Lemma 5. *Let G be the bipartite graph G, and let A be the bipartite adjacency matrix of G. Then,*

$$Per(A) = \#PM(G)$$

Proof. We show that, for any σ, $\prod_{i=1}^{n} A_{i,\sigma(i)} = 1$ if and only if σ gives a perfect matching in G. By definition, $A_{i,\sigma(i)} = 1 \iff (i, \sigma(i)) \in E$. Since the entries are Boolean, and the product is 1, it must be the case that if $\prod_{i=1}^{n} A_{i,\sigma(i)} = 1$ then all edges $A_{i,\sigma(i)}$ are present in the graph. Since σ is a permutation, it must provide an edge $(i, \sigma(i)$ for each vertex $i \in [n]$. Conversely, let S be a perfect matching. It must provide an edge for every $i \in [n]$, and since the perfect matching does not allow any vertex to be covered for more than once, and covers every vertex, we can define a permutation σ such that $\sigma(i) = j$ if $(i, j) \in E$. By our observation, $A_{i,\sigma(i)} = 1$ as well. Hence the proof. □

Note that the problem of testing whether there exist a perfect matching or not can be done in polynomial time, by using Floyd-Warshall flow algorithm. Thus testing whether perfect matching is zero or not can be done in polynomial time. But what about computing the value of the permanent function exactly? Is this in #P at least? We design a non-deterministic Turing machine: given the matrix A, guess the permutation σ, and check if $\prod_{i=1}^{n} A_{i,\sigma(i)} = 1$, and if so accept and reject otherwise. The number of accepting paths is precisely the number of permutations which contributes 1 to the $Per(A)$. Hence $Per \in \#\mathsf{P}$. Indeed, one can also see this by observing that counting the number of perfect matchings in a bipartite graph is in #P (guess a subset of edges and check if it is a perfect matching). We will show soon that Per is #P-complete too.

Before we end this lecture, let us talk about matrices with non-Boolean entries. Let us say $A \in \mathbb{Z}^{n \times n}$. Is the permanent function in #P? An obvious difficulty seems to be that the value of the function can be negative, since the entries could be negative, and it does not make sense to ask for a machine to have negative number of accepting paths. Thus our condition is too stringent, we should relax and ask for can we compute permanent with an oracle access to #P. We will address these in detail in the next class where we introduce a similar combinatorial interpretation of permanent and use that to argue the $\mathsf{FP}^{\#\mathsf{P}}$ upper bound for computing permanent of integer matrices.