PRINCETON UNIVERSITY COS 522: COMPUTATIONAL COMPLEXITY

Lecture 7: Counting classes

Lecturer: *Sanjeev Arora* Scribe:*Manoj*

First we define a few interesting problems:

Given a boolean function ϕ, #SAT is the problem of finding the number of satisfying assignments for ϕ. Given a graph G and two nodes s and t, #PATH is the problem of finding the number of simple paths from s to t in G. #CYCLE is the problem of finding the number of cycles in a given graph.

To study functions (rather than languages) which can be computed "efficiently" we make the following definition.

DEFINITION 1 **FP** *is the set of all functions* $f : \{0,1\}^* \rightarrow \mathbf{N}$ *computable in polynomial time.*

We would like to know if the problems defined above are in **FP**. The next theorem shows that counting problems may be harder than their corresponding decision problems, since it shows that #CYCLE is **NP**-hard, even though deciding whether a graph has a cycle is easy.

THEOREM 1
If #CYCLE $\in$ **FP**, *then* $\mathbf{P} = \mathbf{NP}$.

PROOF: We reduce HAMCYCLE (the decision problem of whether a digraph has a Hamiltonian cycle or not) to #CYCLE. Given a graph G with n nodes in the HAMCYCLE problem, we construct a graph G' for #CYCLE such that G has a HAMCYCLE iff G' has at least n^{n^2} cycles.

Each edge (u, v) in G is replaced by a gadget as shown in Figure 1. The gadget has $N = n \log_2 n$ levels. If G has a Hamiltonian cycle, the G' has at least $(2^N)^n$ cycles, because for each edge from u to v in G, there are 2^N paths from u' to v' in G', and there are n edges in the Hamiltonian cycle in G. On the other hand, if G has no Hamiltonian cycle, the longest cycle in G is of length at most $n - 1$. Also, the number of cycles is bounded above by n^{n-1}. Note that any cycle in G' corresponds to a cycle in G. So G' can have at most $(2^N)^{n-1} \times n^{n-1} < (2^N)^n$ cycles. $\square$

The following class characterizes many counting problems of interest.

DEFINITION 2 #**P** *is the set of all functions* $f : \{0,1\}^* \rightarrow \mathbf{N}$ *such that there is a polynomial time TM* M *and a constant* c *such that*

$$\forall x, f(x) = |\{y : |y| \leq |x|^c \text{ and } M(x, y) \text{ accepts}\}|$$

For example, #CYCLE $\in$ #**P**, because one can construct a polynomial time TM to verify a canonical representation of a cycle in a graph.

DEFINITION 3 *A function* $f \in$ #**P** *is* #**P**-complete *if* $\forall g \in$ #**P**, $g \in \mathbf{FP}^f$

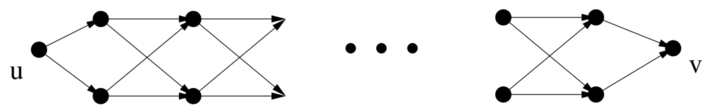

Figure 1: Reducing HAMCYCLE to #CYCLE

Counting versions of **NP**-complete languages naturally lead to #**P**-complete languages. This follows from the way reductions preserve the number of certificates. For instance, the Cook-Levin reduction of an **NP** language to SAT gives a one-one correspondence between the satisfying assignments of the boolean function produced and the accepting tableaux of the **NP**-machine.

Theorem 2
#SAT*,* #CLIQUE *etc are #**P**-complete*

Now we study another problem. The *permanent* of an $n \times n$ matrix A is defined as

$$\text{perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i\sigma(i)} \tag{1}$$

where S_n denotes the set of all permutations of n elements. (Recall that the expression for the determinant is similar, but it involves an additional "sign" term.)

Every $n \times n$ 0-1 matrix A corresponds to a bipartite graph $G(X, Y, \Delta)$, with $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_n\}$ and $\{x_i, y_j\} \in \Delta$ iff $A_{ij} = 1$. Then perm(A) is the number of perfect matchings in G. Another useful way to look at a matrix A is as the adjacency matrix of an n-node digraph (with possible self loops); then perm(A) is the number of cycle covers of A. (A *cycle cover* is a subgraph in which each node has in-degree and out-degree 1; such a subgraph must be composed of cycles.)

For a 0/1 matrix $\text{perm}(A) = \left|\left\{\sigma : \prod_{i=1}^{n} a_{i\sigma(i)} = 1\right\}\right|$, which reveals that the perm is in #**P** for this case. If A is a $\{-1, 0, 1\}$ matrix, then finding the permanent is in $\mathbf{P}^{\#\text{SAT}}$, because then $\text{perm}(A) = \left|\left\{\sigma : \prod_{i=1}^{n} a_{i\sigma(i)} = 1\right\}\right| - \left|\left\{\sigma : \prod_{i=1}^{n} a_{i\sigma(i)} = -1\right\}\right|$. One can show for general integer matrices that computing the permanent is in $\mathbf{FP}^{\#\text{SAT}}$.

The next theorem came as a surprise to researchers in the 1970s, since the permanent seems quite similar to the determinant, which is easy to compute.

Theorem 3 (Valiant)
PERM *is #**P**-complete*

As a warm-up for the proof, we first use an example.

Example 1 Consider the graph in Figure 2. (Unmarked edges have unit weight. We follow this convention through out this lecture). Even without knowing what the subgraph G' is, we can conclude that the permanent of the whole graph is 0, because for each cycle cover in G' there are exactly two cycle covers for the three nodes, one with weight 1 and one with weight -1 (and any non-zero weight cycle cover of the whole graph is composed of a cycle cover for G' and one of these two cycle covers).

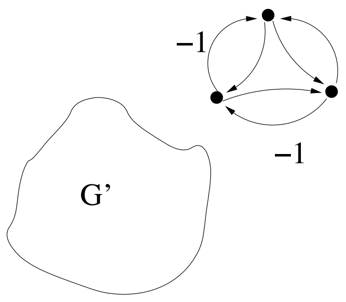

Figure 2: This graph has permanent 0

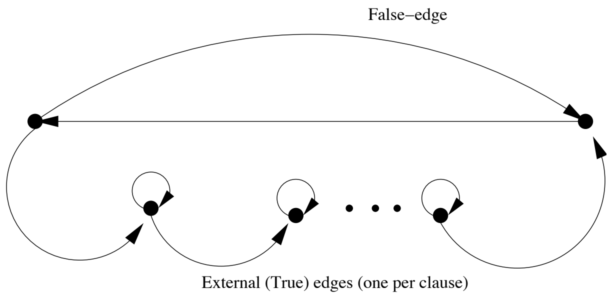

Figure 3: The Variable-gadget

Now we prove Valiant’s Theorem.

Proof: We shall reduce the #**P**-complete problem #3SAT to PERM. Given a boolean formula ϕ with n variables and m clauses, first we shall show how to construct an integer matrix A' with negative entries such that $\text{perm}(A') = 4^{3m}(\#\phi)$. ($\#\phi$ stands for the number of satisfying assignments of ϕ). Later we shall show how to to get a 0-1 matrix A from A' such that knowing $\text{perm}(A)$ allows us to compute $\text{perm}(A')$.

The main idea is that there are two kinds of cycle covers in the digraph G' associated with A': those that correspond to satisfying assignments (we will make this precise) and those that don’t. Recall that $\text{perm}(A')$ is the sum of weights of all cycle covers of the associated digraph, where the weight of a cycle cover is the product of all edge weights in it. Since A' has negative entries, some of these cycle covers may have negative weight. Cycle covers of negative weight are crucial in the reduction, since they help cancel out contributions from cycle covers that do not correspond to satisfying assignments. (The reasoning to prove this will be similar to that in Example 1.) On the other hand, each satisfying assignment contributes 4^{3m} to $\text{perm}(A')$, so $\text{perm}(A') = 4^{3m} \cdot (\#\phi)$.

To construct G' from ϕ, we use three kinds of gadgets as shown in Figures 3, 4 and 5. There is a variable gadget per variable and a clause gadget per clause. There are two possible cycle covers of a variable gadget, corresponding to an assignment of 0 or 1 to that variable. Assigning 1 corresponds to a single cycle taking all the external edges (“true-edges”), and assigning 0 correspond to taking all the self-loops and taking the “false-edge”. Each external edge of a variable is associated with a clause in which the variable appears.

The clause gadget is such that the only possible cycle covers exclude at least one external edge. Also for a given (proper) subset of external edges used there is a unique cycle cover

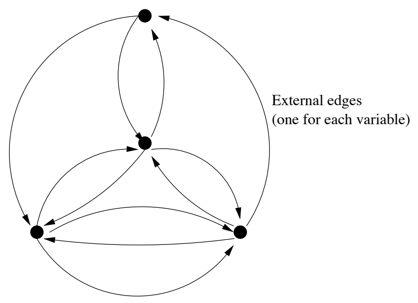

Figure 4: The Clause-gadget

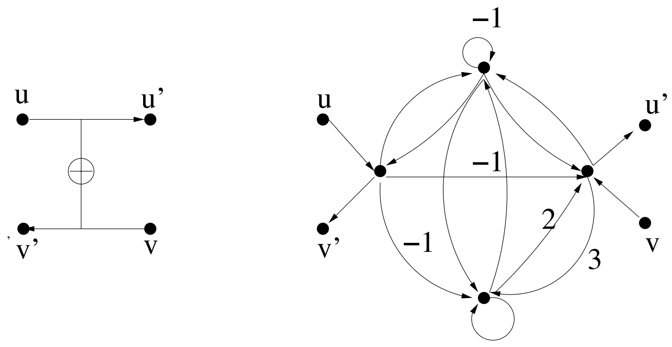

Figure 5: The XOR-gadget

(of weight 1). Each external edge is associated with a variable appearing in the clause.

We will also use a graph called the XOR gadget (Figure 5) which has the following purpose: we want to ensure that exactly one of the edges uu' and vv' (see the schematic representation in Figure 5) is used in a cycle cover that contributes to the total count. So after inserting the gadget, we want to count only those cycle covers which either enter the gadget at u and leave it at u' or enter it at v and leave it at v'. This is exactly what the gadget guarantees: one can check that the following cycle covers have total weight of 0: those that do not enter or leave the gadget; those that enter at u and leave at v', or those that enter at v and leave at u'. In other words, the only cycle covers that have a nonzero contribution are those that either (a) enter at u and leave at u' (which we refer to as using "edge" uu') or (b) enter at v and leave at v' (referred to as using edge vv'). These are cycle covers in the "schematic graph" (which has edges as shown in Figure 5) which *respect* the XOR gadget.

The XOR gadgets are used to connect the variable gadgets to the corresponding clause gadgets so that only cycle covers corresponding to a satisfying assignment need be counted towards the total number of cycle covers. Consider a clause, and a variable appearing in it. Each has an external edge corresponding to the other, connected by an XOR gadget (figure 6). If the external edge in the clause is not taken (and XOR is respected) the external edge in the variable must be taken (and the variable is true). Since at least one external edge of each clause gadget has to be omitted, each cycle cover respecting all the XOR gadgets corresponds to a satisfying assignment. (If the XOR is not respected, we need not count

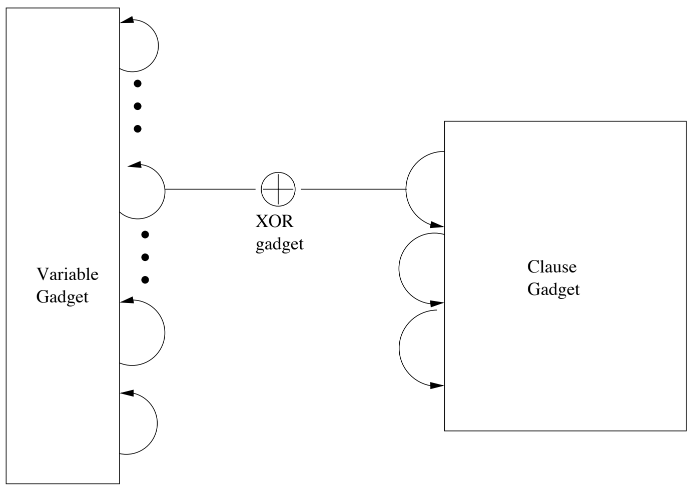

Figure 6: For each clause and variable appearing in it, an XOR-gadget connects the corresponding external edges. There are $3m$ such connections in total.

such a cycle cover as its weight will be cancelled by another cover, as we argued above). Conversely, for each satisfying assignment, there is a cycle cover (unique, in the schematic graph) which respects all the XOR gadgets.

Now, consider a satisfying assignment and the corresponding cycle cover in the schematic graph. Passing (exactly one of) the external edges through the XOR gadget multiplies the weight of each such cover by 4. Since there are $3m$ XOR gadgets, corresponding to each satisfying assignment there are cycle covers with a total weight of 4^{3m} (and all other cycle covers total to 0). So $\text{perm}(G') = 4^{3m}\#\phi$.

Finally we have to reduce finding $\text{perm}(G')$ to finding $\text{perm}(G)$, where G is an unweighted graph. First we reduce it to finding $\text{perm}(G)$ modulo $2^N + 1$ for a large enough N (but still polynomial in $|G'|$). For this, we can replace -1 edges with edges of weight 2^N, which can be converted to N edges of weight 2 in series. Changing edges of (small) positive integral weights (i.e., multiple or parallel edges) to unweighted edges is as follows: cut each (repeated) edge into two and insert a node to connect them; add a self loop to the node. This does not change the permanent, and the new graph has only unweighted edges. □

Exercises

§1 Show that computing the permanent for matrices with integer entries is in $\mathbf{FP}^{\#\text{SAT}}$.