Lecture 4

Complexity classes of functions, #P, and the Permanent

April 10, 2008
Lecturer: Paul Beame
Notes:

We will now define some complexity classes of functions on input strings that output numerical values of strings rather than decision problems where the output is a single bit.

Definition 4.1 The class FP is the set of all functions $f : \{0,1\}^* \to \{0,1\}^*$ (or alternatively, $f : \{0,1\}^* \to \mathbb{N}$) that are computable by a TM with a separate output tape in polynomial time. The class FNP is the analogous class for nondeterministic polynomial time where the requirement for a nondeterministic TM to compute a function is that on a given input x the content of the output tape is $f(x)$ in all accepting computations.

Among the algorithmic problems representable by such functions are "counting problems" related to common decision problems. For example,

Definition 4.2 Let #3-SAT be the problem that takes as input the encoding $\langle\varphi\rangle$ of a 3-CNF formula φ and outputs the number of satisfying assignments of φ.

More generally, given an NP language A defined by $x \in A \Leftrightarrow \exists y \in \{0,1\}^{q(|x|)} R(x,y)$ for a natural polynomial-time computable predicate R associated with A, $\#A$ will be the function mapping x to $\#\{y \in \{0,1\}^{q(|x|)} \mid R(x,y)\}$. Note that this is not a precisely specified since there may be many different choices of R that will work for the same language A. For many natural problems, however, the right choice of the relation R will be obvious.

Definition 4.3 Given a complexity class C, we define the complexity class #C to be the set of all functions $f : \{0,1\}^* \to \mathbb{N}$ such that there is an $R \in$ C and a polynomial q such that $f(x) = \#\{y \in \{0,1\}^{q(|x|)} \mid (x,y) \in R\}$.

In particular, #P is the set of all functions $f : \{0,1\}^* \to \mathbb{N}$ such that there is an $R \in$ P and a polynomial q such that $f(x) = \#\{y \in \{0,1\}^{q(|x|)} \mid (x,y) \in R\}$.

4.1 Function classes and oracles

Let us consider the use of oracle TMs in the context of function classes. For a language A, let FP^A be the set of functions $f : \{0,1\}^* \to \{0,1\}^*$ that can be computed in polynomial time by an oracle TM $M^?$ that has A as an oracle.

Similarly, we can define oracle TMs $M^?$ that allow functions as oracles rather than sets. In this case, rather than receiving the answer from the oracle by entering one of two states, the machine can receive a binary encoded version of the oracle answer on an oracle answer tape. Thus for functions $f : \{0,1\}^* \to \{0,1\}^*$ or $f : \{0,1\}^* \to \mathbb{N}$ and a complexity class C for which it makes sense to define oracle versions, we can define C^f. For a complexity class FC' of functions we can define $\mathsf{C}^{\mathsf{FC}'} = \bigcup_{f \in \mathsf{FC}'} \mathsf{C}^f$.

Note that although $\#\mathsf{P}$ is a class of functions it is very closely related to the unbounded-error class PP. In fact, their closure under polynomial-time Turing reductions is the same.

Theorem 4.4 $\mathsf{P}^{\mathsf{PP}} = \mathsf{P}^{\#\mathsf{P}}$.

Proof There are two directions. One is easy. For $L \in \mathsf{P}^{\mathsf{PP}}$ there is some $A \in \mathsf{PP}$ such that $L \in \mathsf{P}^A$. Furthermore there is some polynomial-time Turing machine M and polynomial p such that $x \in A$ if and only if $\#\{r \in \{0,1\}^{p(|x|)} \mid M(x,r) = 1\} > 2^{p(|x|)} - 1$. The $\mathsf{P}^{\#\mathsf{P}}$ algorithm will simply replace each call A to a call to the function $f \in \#\mathsf{P}$ that computes $\#\{r \in \{0,1\}^{p(|x|)} \mid M(x,r) = 1\}$ and then compares $f(x)$ to $2^{p(|x|)} - 1$.

For the other direction, one has to use a polynomial number of calls to an appropriate PP oracle to replace a call to the $\#\mathsf{P}$ oracle f that on input x returns $f(x) = \#\{y \in \{0,1\}^m \mid M(x,y) = 1\}$ where $m = q(|X|)$ for some polynomial-time computable M. We can denote M' be the TM that on input (z,y) where $y, z \in \{0,1\}^m$ accepts if $y \prec z$ in the lexicographic order. Define M that on input (x,z) flips $m+1$ bits *by* where $b \in \{0,1\}$. If $b = 0$ then run $M(x,y)$. If $b = 1$ then run $M'(z,y)$. The probability of acceptance of M'' is $(N_z + f(x))/2^{m+1}$ where z is the binary representation of integer $N_z \in \{0, \ldots, 2^m - 1\}$. If this is strictly larger than $1/2$ then we know that $f(x) > 2^m - N_z$. We can query this language with $m+1$ different values of z to do a binary search for the value of $f(z)$. □

Definition 4.5 A function f is $\#\mathsf{P}$-complete iff

1. $f \in \#\mathsf{P}$.
2. For all $g \in \#\mathsf{P}$ we have $g \in \mathsf{FP}^f$.

As 3-SAT is NP-complete, #3-SAT is $\#\mathsf{P}$-complete:

Theorem 4.6 #3-SAT is $\#\mathsf{P}$-complete.

Proof The reduction produced by the Cook-Levin tableau is "parsimonious", in that it preserves the number of solutions. More precisely, in circuit form there is precisely one satisfying assignment for the circuit for each NP witness y. Moreover, the conversion of the circuit to 3-SAT enforces precisely one satisfying assignment for each of the extension variables associated with each gate. □

Since the standard reductions are frequently parsimonious, they can be used to prove $\#\mathsf{P}$-completeness of many counting problems relating to NP-complete problems. In some instances they are not parsimonious but can be made parsimonious. For example we have the following.

Theorem 4.7 $\#HAM\text{-}CYCLE$ is #P-complete.

The set of #P-complete problems is not restricted to the counting versions of NP-complete problems, however; interestingly, problems in P can have #P-complete counting problems as well.

Consider $\#CYCLE$, the problem of finding the number of directed simple cycles in a graph G. (The corresponding problem $CYCLE$ is in P).

Theorem 4.8 $\#CYCLE$ is #P-complete.

Proof We reduce from $\#HAM\text{-}CYCLE$. We will map the input graph G for $\#HAM\text{-}CYCLE$ to a graph G' for $\#CYCLE$. Say G has n vertices. G' will have a copy u' of each vertex $u \in V(G)$, and for each edge $(u, v) \in E(G)$ the gadget in Figure 4.1 will be added between u' and v' in G'. This gadget consists of $N = n\lceil \log_2 n \rceil + 1$ layers of pairs of vertices, connected to u' and v' and connected by $4N$ edges within. The number of paths from u' to v' in G' is $2^N \geq 2n^n$. Each simple cycle of length ℓ in G yields $(2^N)^\ell = 2^{N\ell}$ simple cycles in G'. If G has k Hamiltonian cycles, there will be $k2^{Nn}$ corresponding simple cycles in G'. Now G has feweer than n^n simple cycles of any length, in particular of length $\leq n-1$. The total number of simple cycles in G' corresponding to these cycles of length $\leq n-1$ is $< n^n 2^{N(n-1)} = 2^{Nn-1}$ since $n^n \leq 2^{N-1}$. Therefore we compute $\#HAM\text{-}CYCLE(G) = \lfloor \#CYCLE(G')/2^{Nn} \rfloor$. □

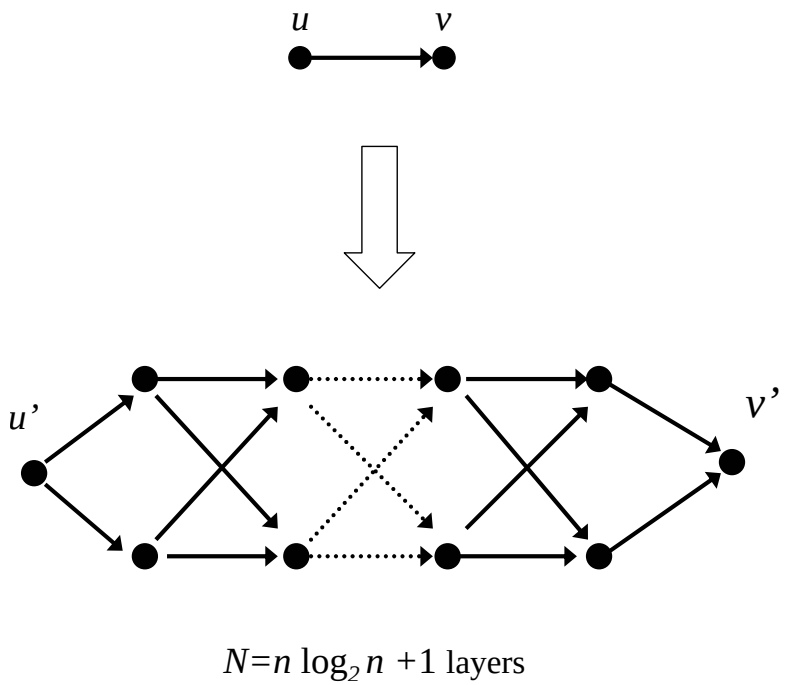

Figure 4.1: Edge replacement in $\#HAM\text{-}CYCLE$ to $\#CYCLE$ reduction.

The following corollary is left as an exercise:

Corollary 4.9 $\#2\text{-}SAT$ is #P-complete.

4.2 Determinant and Permanent

Some interesting problems in matrix algebra Given an $n \times n$ matrix $A = (a_{ij})$, the determinant of A is

$$det(A) = \sum_{\sigma \in S_n} (-1)^{sgn(\sigma)} \prod_{i=1}^{n} a_{i\sigma(i)},$$

where S_n is the set of permutations of $[n] = \{1, \ldots, n\}$ and $sgn(\sigma)$ is the is the number of transpositions required to produce σ modulo 2. This problem is in FP.

The $(-1)^{sgn(\sigma)}$ is apparently a complicating factor in the definition of $det(A)$, but if we remove it we will see that the problem actually becomes harder. This is called the *permanent* of matrix A:

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

Let $PERM$ be the problem of computing the permanent of a matrix. and 0-1$PERM$ the problem in the case that the matrix has binary entries. We can view the matrix A as the weighted adjacency matrix of a bipartite graph on $[n] \times [n]$. Each $\sigma \in S_n$ corresponds to a perfect matching in this graph. If we view the weight of a matching as the product of the weights of its edges the permanent is the total weight of all matchings in the graph.

In particular a 0-1 matrix A corresponds to an unweighted bipartite graph G for which A is the adjacency matrix, and $perm(A)$ represents the number of perfect matchings on G. Let $\#BIPARTITE\text{-}MATCHING$ be the problem of counting all such matchings. Therefore 0-1$PERM = \#BIPARTITE\text{-}MATCHING \in \#\mathsf{P}$.

Alternatively, an $n \times n$ matrix A can be viewed as a weighted adjacency matrix of a directed graph G on n vertices (with possibly self-loops). Now each permutation $\sigma \in S_n$ can be decomposed into a union of disjoint cycles. For example, if $\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 2 & 1 & 6 \end{pmatrix} \in S_6$ then σ can also be written in cycle form as (1 3 5)(2 4)(6) where the notation implies that each number in the group maps to the next and the last maps to the first. These cycles cover all of the points $[n]$. For a directed graph G, a *cycle-cover of* G is a union of simple cycles of G that contains each vertex precisely once. In particular the edges corresponding to a term $\prod_{i=1}^{n} a_{i\sigma(i)}$ is the permanent of A corresponds to the product of the weights of edges in the directed graph G corresponding to the cycle-cover corresponding to σ, which we can view as the weight of the cycle-cover. Therefore $perm(A)$ is the total weight of all cycle-covers of G.

For a weighted, directed graph G, define $PERM(G)$ as the total weight of all cycle-covers of G, where the weight of a cycle-cover is the product of the weights of all its edges. Thus, for an unweighted graph G, $PERM(G)$ is the number of cycle-covers of G. The hardness of 0-1$PERM$ is established by showing that the problem of finding the number of cycle-covers of G is hard.

Theorem 4.10 (Valiant) 0-1$PERM$ is $\#\mathsf{P}$-complete.

Proof We will reduce $\#3\text{-}SAT$ to 0-1$PERM$ in two steps. Given any 3-SAT formula ϕ, in the first step, we will create a weighted directed graph G' (with small weights) such that

$$PERM(G') = 4^{3m} \#(\varphi)$$

where m is the number of clauses in φ. In second step, we will convert G' to an unweighted graph G such that $PERM(G') = PERM(G) \mod M$, where M will only have polynomially many bits.

First, we will construct G' from ϕ. The construction will be via gadgets. The VARIABLE gadget is shown in Figure 4.2. All the edges have unit weights. Notice that it contains one dotted edge for every occurrence of the variable in ϕ. Each dotted edge will be replaced by a subgraph which will be described later. Any cycle-cover either contains all dotted edges of positive occurrence (and all self-loops of negative occurrence) or vice versa.

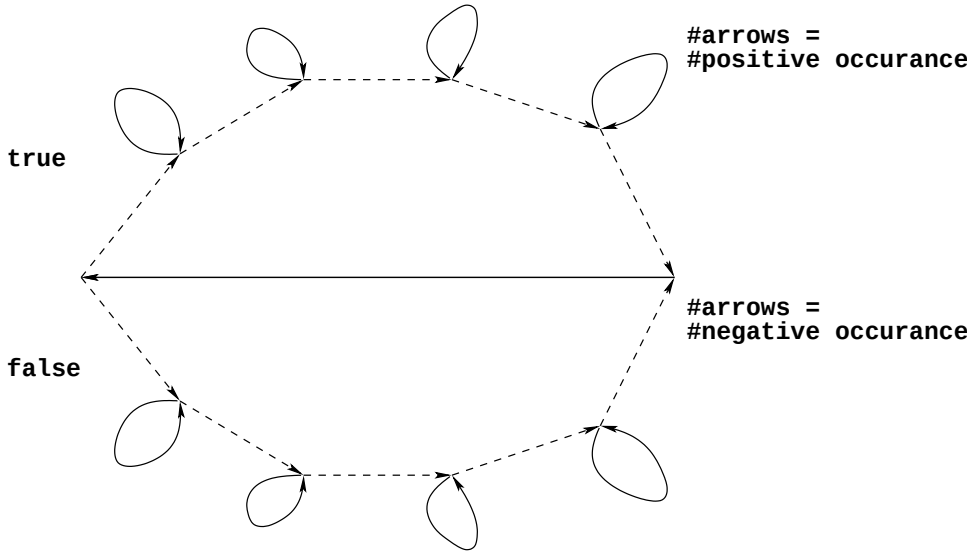

Figure 4.2: The VARIABLE gadget

The CLAUSE gadget is shown in Figure 4.2. It contains three dotted edges corresponding to three variables that occur in that clause. All the edges have unit weights. This gadget has the property that in any cycle-cover, at least one of the dotted edges is not used.

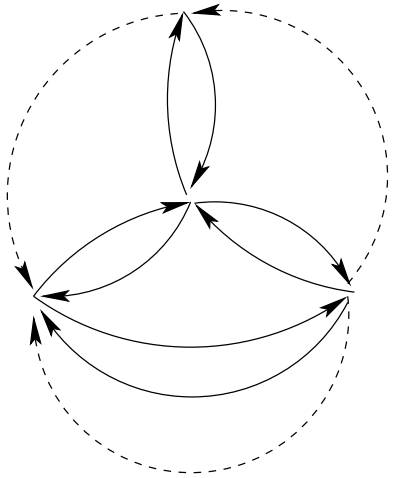

Figure 4.3: The CLAUSE gadget

Now, given any clause C and any variable x contained in it, there is a dotted edge (u, u') in the CLAUSE gadget for the variable and a dotted edge (v, v') in the VARIABLE gadget for the clause. These two dotted edges are replaced by an XOR gadget shown in Figure 4.2.

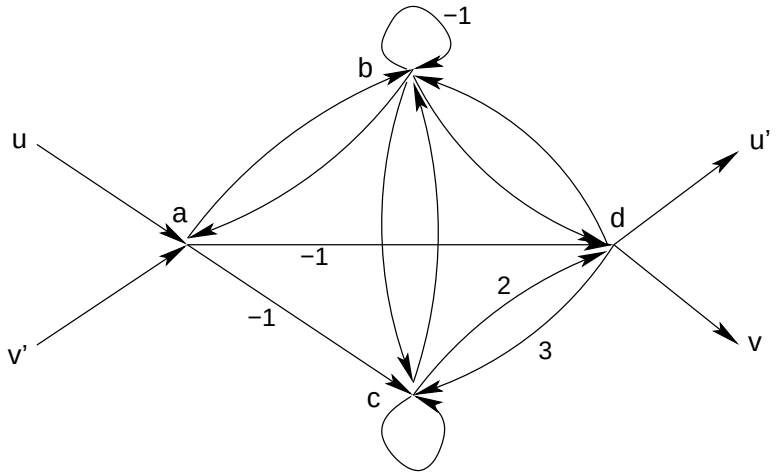

Figure 4.4: The CLAUSE gadget

The XOR gadget has the property that the total contribution of all cycle-covers using none or both of (u, u') and (v, v') is 0. For cycle-covers using exactly one of the two, the gadget contributes a factor of 4. To see this, lets consider all possibilities:

1. None of the external edges are present: The cycle-covers are [a c b d], [a b][c d], [a d b][c] and [a d c b]. The net contribution is (-2) + 6+ (-1) + (-3) = 0.

2. (u, a) and (a, v') are present: The cycle-covers are [b c d], [b d c], [c d][b] and [c][b d]. The net contribution is (2) + (3)+ (-6) + (1) = 0.

3. (v, d) and (d, u') are present: The cycle-covers are [a b][c] and [a c b]. The net contribution is 1 + (-1) = 0.

4. All four external edges are present: The cycle-covers are [b c] and [b][c]. The net contribution is 1 + (-1) = 0.

5. (v, d) and (a, v') are present: The cycle-covers are [d b a][c] and [d c b a]. The net contribution is 1 + 3 = 4.

6. (u, a) and (d, v') are present: The cycle-covers are [a d][b c], [a d][b][c], [a b d][c], [a c d][b], [a b c d] and [a c b d]. The net contribution is (-1) + 1 + 1 + 2 + 2 + (-1) = 4.

There are $3m$ XOR gadgets. As a result, every satisfying assignment of truth values to ϕ will contribute 4^{3m} to the cycle-cover and every other assignment will contribute 0. Hence,

$$PERM(G') = 4^{3m}\#(\phi)$$

Now, we will convert G' to an unweighted graph G. Observe that $PERM(G') \leq 4^{3m}2^n \leq 2^{6m+n}$. Let $N = 6m + n$ and $M = 2^N + 1$. Replace the weighted edges in G' with a set of unweighted

edges as shown in Figure 4.2. For weights 2 and 3, the conversion does not affect the total weight of cycle-covers. For weight -1, the conversion blows up the total weight by $2^N \equiv -1(mod M)$. As a result, if G is the resulting unweighted graph, PERM(G') = PERM(G) mod M.

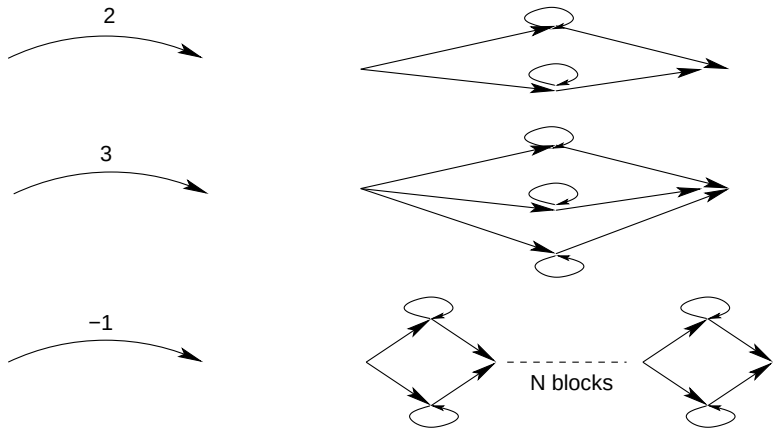

Figure 4.5: The VARIABLE gadget

Thus, we have shown a reduction of #3-SAT to 0-1PERM. This proves the theorem.

□