CS294 Partition Functions **Fall 2023**

Lecture Note 2

Instructor: Alistair Sinclair

2.1 #P-Completeness

As mentioned in the previous lecture, counting problems can be viewed as a natural generalization of decision problems, in which the goal is to count the number of solutions rather than just to determine whether at least one solution exists. Valiant [Val79a, Val79b] introduced the class #P to formally capture this notion.

Definition 2.1. *Let Σ be a finite alphabet. A function $f : \Sigma^* \to \mathbb{N}$* belongs to #P *if there is a polynomial time non-deterministic Turing machine M s.t., for any input $x \in \Sigma^*$, the number of accepting computations of M on x is exactly $f(x)$.*

All "natural" problems in NP have a counting version in #P. For example, to see that #SAT (counting satisfying assignments of a boolean formula φ) is in #P, we simply design a Turing machine that non-deterministically checks every possible assignment and accepts if the assignment satisfies φ.

Definition 2.2. *A function $f : \Sigma^* \to \mathbb{N}$ is* #P-hard *if the computation of any function in #P is polynomial time reducible to the computation of f. If in addition f belongs to #P then f is* #P-complete.

Again, it is easy to see that the counting versions of most NP-complete decision problems are #P-complete. This follows from the fact that most reductions proving NP-completeness are (or can be modified to be) *parsimonious*, in the sense that they preserve the number of solutions (possibly up to some easily computed factor). This applies in particular to Cook's generic reduction from any problem in NP to SAT, establishing that #SAT is #P-complete. #P-completeness for other problems follows by parsimonious reduction from #SAT.

2.2 The permanent

#P-completeness starts to become interesting when we observe that the counting versions of many decision problems in P are also #P-complete! The first major result of this kind is also due to Valiant [Val79a], who proved that counting the number of perfect matchings in a bipartite graph is #P-complete. Note that deciding whether a given bipartite graph has a perfect matching is well known to be in P, e.g., via a reduction to network flow. The counting problem here is also of wider interest because it is equivalent to computing the *permanent* of the adjacency matrix of the graph. The permanent of an $n \times n$ matrix A is defined as

$$\mathrm{per}(A) := \sum_{\sigma} \prod_{i=1}^{n} a_{i,\sigma(i)}, \tag{2.1}$$

where the sum is over all permutations σ of $[n]$. Note that this formula is syntactically very similar to that of the determinant, the only difference being that in the determinant each term includes the sign of the permutation σ.

Exercise: Verify that, for a bipartite graph $G = (V_1, V_2, E)$ with $|V_1| = |V_2| = n$, the number of perfect matchings in G is equal to $\mathrm{per}(A(G)$, where $A(G)$ is the bipartite adjacency matrix of G (with rows, columns indexed by V_1, V_2 respectively).

Theorem 2.3 ([Val79a]). *Computing the permanent of a matrix A with 0-1 entries is #P-complete.*

We present a proof that is slightly simpler than Valiant's original one, and is usually attributed to Papadimitriou and Arora/Barak.

Proof. Evidently the problem is in #P: $\mathrm{per}(A)$ is just the number of non-zero terms in (2.1), and a non-deterministic Turing machine can check each permutation σ and accept iff the corresponding term is non-zero.

To prove #P-hardness, we give a reduction from #3-SAT, the problem of counting the number of satisfying assignments of a formula φ in 3-CNF. The reduction uses an alternative graphical characterization of the permanent. Let G_A be the n-vertex *directed* graph (possibly with self-loops) whose adjacency matrix is A. A *cycle cover* of G_A is a set of directed cycles that includes each vertex exactly once. Then it is easy to see [**Exercise!**] that $\mathrm{per}(A)$ is equal to the number of cycle covers of G_A. Thus, given φ, it suffices to construct a directed graph G_φ such that the number of cycle covers of G_φ is equal to (some easily computed factor c times) the number of satisfying assignments of φ.

We construct G_φ using three types of gadget: a variable gadget, a clause gadget, and an XOR gadget. (See figure 2.1.) The edges of the XOR gadget are labeled with integer weights (some of which are negative); all unlabeled edges have weight 1. We will show that the sum of the weights of all cycle covers of G_φ is equal to 4^{3m} times $\#\mathrm{SAT}(\varphi)$, where the weight of a cycle cover is the product of the weights of its edges and m is the number of clauses in φ.

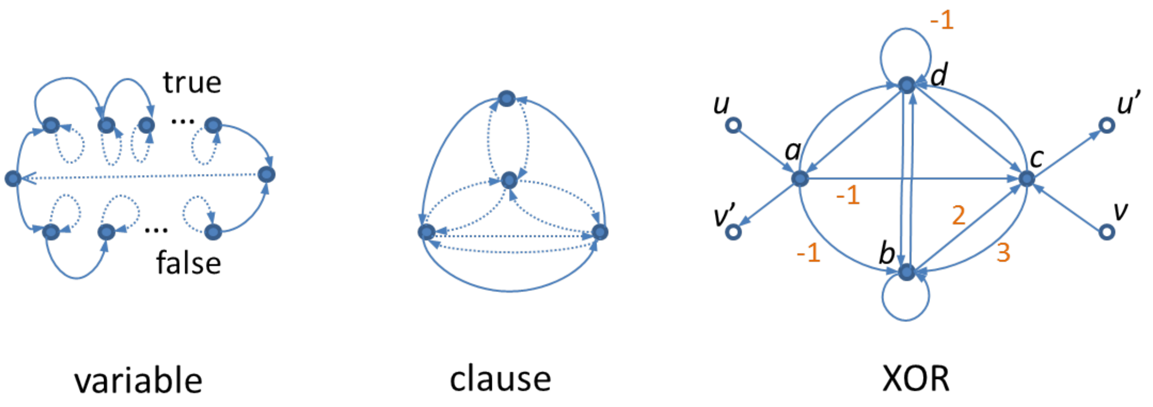

Figure 2.1: Gadgets

The key properties of these gadgets are the following. (In the variable and clause gadgets, we refer to the solid edges as "external" and the dotted ones as "internal".)

variable: This gadget has exactly two cycle covers, one of which takes the external edges on the "true" side and the self loops on the "false" side, and vice versa. The length of each of the true/false chains is equal to the number of clauses in which the variable appears. [The choice of cycle cover will correspond to setting the value of this variable to true or false.]

clause: This gadget has exactly one cycle cover that omits each non-empty subset of the external edges (and no cycle cover that includes all the external edges). [The omitted edges will correspond to the literals that are satisfied in this clause, of which there must be at least one.]

XOR: This is the most subtle gadget, and has only two families of cycle covers with non-zero combined weight: those that include the external edges from u and to u', and those that include the external edges from v and to v'. Each of these families has total weight 4. These properties can be verified by writing the adjacency matrix of the gadget as

$$B = \begin{pmatrix} 0 & -1 & -1 & 1 \\ 0 & 1 & 2 & 1 \\ 0 & 3 & 0 & 1 \\ 1 & 1 & 1 & -1 \end{pmatrix}$$

where the rows/columns of B are enumerated as a, b, c, d in the diagram of the gadget. Now one can readily check that

$$\begin{aligned} \mathrm{per}(B) = \mathrm{per}(B_{a,a}) = \mathrm{per}(B_{c,c}) = \mathrm{per}(B_{ac,ac}) &= 0\,; \\ \mathrm{per}(B_{a,c}) = \mathrm{per}(B_{c,a}) &= 4\,, \end{aligned}$$

where $B_{i,j}$ denotes the submatrix of B obtained by deleting the ith row and jth column. Note in particular how the negative matrix entries ensure that unwanted cycle covers cancel out. [This gadget functions as an XOR, in the sense that, when inserted into the graph to replace edges (u, u') and (v, v'), it gives a non-zero contribution only to cycle covers that include exactly one of these edges. This will be used to ensure consistency between the variable and clause gadgets.]

Exercise: Verify the above properties of per(B) and its submatrices, and that these properties imply the desired consequences for the cycle covers of the gadget claimed above.

The entire graph G_φ consists of one variable gadget for each variable of φ, one clause gadget for each clause, and one XOR gadget for each occurrence of a literal in a clause ($3m$ in total), inserted between the corresponding variable and clause gadgets. Figure 2.2 illustrates how this works for one sample clause. Note in particular how the XOR gadgets are used to ensure that, in any cycle cover, the external edges omitted in any clause gadget correspond precisely to those variables whose assignment satisfies the clause. This observation, and the fact that any cycle cover must omit at least one edge from each clause, ensures that each satisfying assignment corresponds to a set of cycle covers of total weight precisely 4^{3m} (note that $3m$ is the total number of XOR gadgets). This completes the reduction.

So far we have shown that computing the permanent of a matrix with small integer values is #P-hard. To complete the proof of the theorem, we need to show that this still holds when the matrix entries are 0-1. Note first that an edge (u, v) of weight $w > 1$ can be replaced by w parallel paths $u \to x_i \to v$ with all edges of weight 1 without changing the value of the permanent. Since the only negative edge weights are -1, this gets us to the case where all non-zero matrix entries are in $\{\pm 1\}$. Now note that the resulting permanent certainly lies in the range $[-n!, n!]$, so since $n! < 2^{n^2}$ it suffices to compute it modulo $2^m + 1$ for $m \approx n^2$. The -1 entries now become 2^m, and can be replaced by m edges of weight 2 in series, each of which can then be replaced by two parallel paths as before, leaving us with a graph with only unit edge weights. These operations cause only a polynomial blow-up in the size of the graph. □

2.3 Other problems

The permanent plays a role in #P-completeness analogous to that played by SAT in NP-completeness, in that many other (non-obvious) problems can be proved #P-complete via reduction from it (see Valiant's

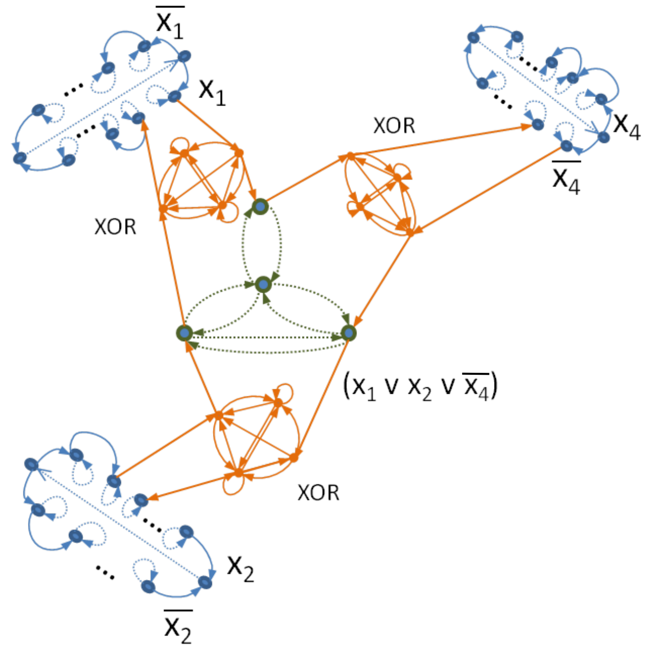

Figure 2.2: Portion of the graph G_φ involving one clause.

paper [Val79b] for several examples). The more interesting of these reductions are not parsimonious, but use other techniques such as polynomial interpolation. We give two illustrative examples of non-trivial #P-hardness reductions.

Let #(s,t)-PATHS be the problem of counting the number of simple paths between vertices s and t in a graph G.

Theorem 2.4. *#(s,t)-PATHS is #P-complete*

Proof. We give a reduction from #HAM, counting Hamilton paths in a graph. (This standard NP-complete problem is easily seen to be #P-complete by parsimonious reduction from #SAT.) Construct graph G' by replacing each edge of G by a "chain of diamonds" of length k. Then each (s,t) path in G of length ℓ corresponds to $2^{k\ell}$ paths of length $k\ell$ in G'. Thus the total number of (s,t) paths in G' is $\sum_\ell p_\ell 2^{k\ell}$, where p_ℓ is the number of such paths of length ℓ in G. Now we choose k large enough so that the final term $p_{n-1}2^{k(n-1)}$ dominates the sum of all the other terms. Specifically, we have (rather crudely) $\sum_{\ell<n-1} p_\ell 2^{k\ell} \le n!2^{k(n-2)} \le 2^{n^2+k(n-2)}$, which is less than $2^{k(n-1)}$ if we take $k > n^2$. With this choice of k, we can read off the top coefficient p_{n-1} from the total number of paths in G' by truncation, giving us the number of Hamilton paths from s to t. □

The above reduction is quite straightforward because it is easy to "boost" the number of structures corresponding to the "hard" coefficient p_{n-1} in the polynomial $Z(\lambda) := \sum_\ell p_\ell \lambda^\ell$. Here is a more sophisticated example that uses the full power of polynomial interpolation.

Let #MATCH be the problem of counting the number of matchings (of all sizes) in a graph G. (Note that the corresponding decision problem here is trivial: every graph contains the empty matching!)

Theorem 2.5. *The problem #MATCH is #P-complete.*

Proof. We give a reduction from counting perfect matchings to #MATCH. Let G be an arbitrary bipartite graph with $2n$ vertices, and for $0 \le k \le n$ let m_k denote the number of matchings in G with exactly k edges. Now let G_s denote the graph obtained by appending to each vertex v on the left side of G s new vertices, each connected only to v. Then the total number of matchings in G_s is $\sum_k m_k(s+1)^{n-k} = (s+1)^n \sum_k m_k(\frac{1}{s+1})^k$. Note that, by varying s, we obtain different points on the matching polynomial $Z_G(\lambda) = \sum_k m_k\lambda^k$ of G. Since Z has degree at most n, if we evaluate it at $n+1$ different points we can use Lagrange interpolation to recover all its coefficients, i.e., the numbers of matchings of every size in G (including, in particular, m_n, the number of perfect matchings). Each such evaluation involves solving the problem #MATCH on a graph G_s, and the interpolation runs in time polynomial in n and the logarithms of the sizes of the numbers involved. This completes the reduction. □

An alternative perspective on #MATCH is provided by looking more closely at the matching polynomial (or monomer-dimer partition function) $Z_G(\lambda)$. Clearly, if we view λ as part of the input, then computing $Z_G(\lambda)$ is #P-hard, since we could evaluate the polynomial at $n+1$ values of λ and use interpolation to extract all the coefficients, including the number of perfect matchings m_n. Moreover, Theorem 2.5 also tells us that it is #P-hard to evaluate $Z_G(\lambda)$ at the point $\lambda = 1$. But what about other fixed values of λ? We can show that this problem is in fact #P-hard at *every* fixed value of λ (except for $\lambda = 0$ when $Z(0)$ is trivially 1). The trick is to modify G so as to simulate different values of λ, as we did in the proof of Theorem 2.5. For the graphs G_s defined there, notice that

$$Z_{G_s}(\lambda) = \sum_k m_k\lambda^k(1+s\lambda)^{n-k} = (1+s\lambda)^n \sum_k m_k \left(\frac{\lambda}{1+s\lambda}\right)^k = (1+s\lambda)^n Z_G\big(\lambda/(1+s\lambda)\big).$$

Thus evaluating the matching polynomial Z_{G_s} at any fixed, non-zero value of λ, for a sequence of values of s, is tantamount to evaluating Z_G at the set of points $\{\frac{\lambda}{1+s\lambda}\}$. This allows us to interpolate and find the coefficients of Z_G. We have proved:

Theorem 2.6. *The problem of computing the matching polynomial $Z_G(\lambda)$ at any fixed value $\lambda \neq 0$ is #P-hard.*

Much more can be said about the complexity of exactly computing partition functions: in particular, for a systematic classification of the complexity of partition functions of spin systems, see [BG05]. The overall message, however, is simple: these problems are #P-hard in all but a very small number of special cases (two of which we'll see in the next lecture). Hence for the rest of the course we will be focusing almost entirely on *approximate* computation.

2.4 #P and Computational Complexity

How does #P relate to more familiar classes such as NP and PSPACE? It should be clear that, if #P $\subseteq$ P (or more correctly, #P $\subseteq$ FP, the class of *functions* computable in polynomial time), then P = NP [why?]; however, the converse implication is not true. Equally it should be clear that #P is contained in PSPACE [why?]. A famous theorem due to Toda [Tod91] shows that the polynomial hierarchy is contained in $\mathrm{P}^{\#\mathrm{P}}$, implying that #P is "more powerful than" any finite level of quantifier alternation. We won't dwell on these topics here as they are not the main concern of this class; we refer the interested reader to the Complexity Theory class, CS278, or the textbook [AB09].

References

[AB09] S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 2009.

[BG05] A. Bulatov and M. Grohe. The complexity of partition functions. *Theoretical Computer Science*, 348:148–186, 2005.

[Tod91] S. Toda. PP is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, 20:865–877, 1991.

[Val79a] L.G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[Val79b] L.G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8:410–421, 1979.