INFR11102: Computational Complexity 07/11/2019

Lecture 16: Counting complexity

Lecturer: Heng Guo

1 Lovász's perfect matching algorithm

Let $G = (V, E)$ be a bipartite graph with two equal parts. Namely, $V = V_1 \cup V_2$ where V_1 and V_2 are disjoint, $|V_1| = |V_2|$, and $E \subseteq V_1 \times V_2$. A perfect matching (PM) is a subset $M \subseteq E$ of edges so that every vertex is adjacent to exactly one edge in M. Alternatively, a perfect matching M can be thought of as a permutation σ_M of $[n]$, so that $(i, \sigma_M(i)) \in E$.

Name: Bi-PM

Input: A bipartite graph G with two equal parts.

Output: Does G have a perfect matching?

Let $|V_1| = |V_2| = n$. Let A be the n-by-n bi-adjacency matrix of G. Rows of A are indexed by vertices in V_1, and columns by vertices in V_2. $A_{u,v} = 1$ if $(u, v) \in E$ and $A_{u,v} = 0$ otherwise. The determinant of A is defined as

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

where S_n is the symemtric group whose elements are permutations of $[n]$. Notice that if $\det(A) \neq 0$, then we know that there must be some non-zero term, and thus G has a perfect matching. However, if $\det(A) = 0$, then it might be because G has no perfect matching, or non-zero terms cancel. We need to figure out which is the case.

To get around this, we associate a variable $x_{u,v}$ to each edge (u, v). In other words, consider A_X, a n-by-n matrix, where $A_{u,v} = x_{u,v}$ if $(u, v) \in E$, and 0 otherwise. Then $\det(A_X)$ becomes a polynomial in variables $(x_{u,v})_{(u,v)\in E}$. Notice that if G has a perfect matching M, then σ_M induces a monomial in $\det(A_X)$, and $\det(A_X)$ is not identically zero. Otherwise if G does not have a perfect matching, then $\det(A_X)$ is identically zero.

Lovász's algorithm [Lov79] is thus to simply test whether $\det(A_X)$ is identically zero, by randomly assigning values to these variables. The algorithm works because of the following lemma, due to Zippel [Zip79] and Schwartz [Sch80].

Lemma 1. *Let $p(x_1, x_2, \cdots, x_n)$ be a non-zero polynomial of degree d. Let S be a finite set of integers. Then, if $a_1, \cdots, a_n$ are randomly chosen from S (with replacement), then*

$$\Pr[p(a_1, \ldots, a_n) \neq 0] \geq 1 - \frac{d}{|S|}.$$

The proof of Lemma 1 can be found in [AB09, Lemma A.36].

The degree of $\det(A_X)$ is at most n. To apply Lemma 1 to Bi-PM, we just randomly assign values from $[4n]$ to $x_{i,j}$, and then evaluate the determinant. (The determinant has exponentially many terms, but it can be evaluated efficiently using, say, Gaussian elimination.) Under this random assignment, by Lemma 1, $\det(A_X) \neq 0$ with probability at least $1 - \frac{n}{4n} = 3/4$ if G has a PM, and $\det(A_X) = 0$ with probability 1 if G does not have a PM. If we want to get better success probability, then we can simply run it again if $\det(A_X) = 0$. After t independent runs, the error probability goes down to 4^{-t}.

The advantage of this algorithm is that $\det(\cdot)$ is not only efficiently computable, it can even be computed efficiently in parallel. It is contained in a complexity class called NC, which captures efficient parallel computation. For more details, see [AB09, Chapter 6.7.1].

You might have noticed that the following quantity, called the *permanent* of the matrix,

$$\operatorname{per}(A) := \sum_{\sigma \in S_n} \prod_{i=1}^{n} A_{i,\sigma(i)}, \tag{1}$$

does not have the annoying cancellation, and $\operatorname{per}(A)$ indeed counts the number of PM's in G. Unfortunately, $\operatorname{per}(A)$ is a quantity intractable to compute, and it relates to counting complexity.

2 Counting complexity

Recall that for NP problems, we want to decide whether there exists a certificate. There are other situations where we are interested in, instead, the number of certificate. It often has something to do with computing the marginal probability of various random variables. The following counting complexity class #P is introduced by Les Valiant [Val79a, Val79b] to capture the apparent intractability of counting problems.

Definition 1. *A function $f : \{0,1\}^* \to \mathbb{N}$ is in #P if there exist a polynomial-time TM M and a polynomial $p(\cdot)$ such that for every $x \in \{0,1\}^*$:*

$$f(x) = \left|\{y \mid |y| \leq p(|x|) \text{ and } M(x,y) = 1\}\right|.$$

To define #P-hard problems, we need the notion of Turing reductions. In Turing's original paper [Tur37], he defined oracle Turing Machines. Let O be a language. We equip a TM M with an oracle O, denoted M^O, if M has a special tape, and during the execution of M, it can write x on the special tape, and ask the oracle to get whether $x \in O$ in one step. Oracle TMs relate to *Turing reductions*. Then we say A is *Turing reducible* to B, denoted $A \leq_{\mathtt{t}} B$, if there exists a TM M with oracle B that computes A. Again, the intuition of this notation is that A is easier than B. The oracle should be thought of as some subroutine that we can invoke.

#P-hardness and #P-completeness are defined with respect to Turing reductions.

Definition 2. *A function f is said to be* #P-hard *if for any function $g \in$ #P, there is a polynomial-time TM M with oracle f that computes g.*

A function f is #P-complete *if f is #P-hard and $f \in$ #P.*

The most natural #P-complete problem is the counting version of Sat.

Name: #Sat

Input: A CNF formula φ.

Output: The number of satisfying assignment of φ.

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

The proof of Theorem 2 is the same as that of the Cook-Levin theorem. One only need to verify that the number of certificates is preserved through the reduction.

It is easy to see from the definition that, counting problems are harder than the corresponding decision problems. Indeed, there are situations where finding a certificate is easy but counting the number is hard. Recall the permanent $\text{per}(A)$ of a matrix A defined in (1).

Valiant [Val79a] showed that $\text{per}(A)$ is #P-complete, even if A is a $\{0,1\}$-matrix. Hence $\text{per}(A)$ is a hard to compute quantity. In fact, the original motivation of introducing the complexity class #P is to capture this apparent intractability of the permanent. As described before, when A is a $\{0,1\}$-matrix, $\text{per}(A)$ counts the number of perfect matchings in the corresponding bipartite graph.

Name: #PM

Input: A graph G.

Output: The number of perfect matchings in G.

Clearly computing $\text{per}(A)$ is a special case of #PM, when the input is restricted to bipartite graphs.

Theorem 3. #PM *is #P-complete.*

We will omit the proof of Theorem 3. Full details can be found in Valiant's original paper [Val79a] or [AB09, Theorem 17.11]. Essentially, the proof is a "local" reduction in which gadgets are built to mimic variables and clauses of a CNF formula. The main reason we omit it is that these gadgets are highly complicated, and Valiant in [Val79a] explained that why smaller gadgets do not exist. Nonetheless, the existence of these gadgets seems rather "coincidental".

Note that, in contrast to Theorem 3, deciding whether PM exists (or finding one if so) can be done in polynomial-time by Edmonds's algorithm [Edm65], even in general graphs.

A matching is a subset of edges so that every vertex is incident to at most one edge. In other words, it is not necessary that all vertices are matched in a matching, unlike in a perfect matching. What we will show, is that counting (not necessarily perfect) matchings is still #P-complete. The reduction will illustrate an important technique of counting reductions, called "interpolation". It also showcases the difference between many-one reductions and Turing reductions.

Name: #Matching

Input: A graph G.

Output: The number of matchings in G.

Theorem 4. #Matching *is* #P*-complete.*

Proof. We construct a Turing reduction #PM $\leq_t$ #Matching. Given an instance $G = (V, E)$ to #PM, we construct a series of new graphs G_t. For each $t \geq 0$, we add t new vertices, $u_1^v, \cdots, u_t^v$ for each $v \in V$, and connect every one of them to v. Thus, $G = G_0$.

Let Z_k be the number of matchings in G of size $n-k$ (we measure the size of a matching by the number of vertices matched). Then the number of perfect matchings is Z_0. The total number of matchings, denoted M_0 is thus

$$M_0 = \sum_{k=0}^{n} Z_k.$$

On the other hand, for each matching of size $n-k$ in G, there are $(t+1)^k$ distinct matchings corresponding to it in G_t, by matching the k unmatched vertices to the t new ones. Thus, in G_t, the number of matchings is

$$M_t = \sum_{k=0}^{n} (t+1)^k Z_k.$$

Thus, we can evaluate M_t by an oracle to #Matching for $t = 0, 1, \cdots, n$. We view (Z_k) as variables, and then these evaluations form a linear system:

$$\begin{aligned}
Z_0 + Z_1 + \cdots + Z_n &= M_0, \\
Z_0 + 2Z_1 + \cdots + 2^n Z_n &= M_1, \\
\vdots \qquad & \quad \vdots \\
Z_0 + (t+1)Z_1 + \cdots + (t+1)^n Z_n &= M_t, \\
\vdots \qquad & \quad \vdots \\
Z_0 + (n+1)Z_1 + \cdots + (n+1)^n Z_n &= M_n.
\end{aligned}$$

The coefficient matrix of the above system is called a Vandermonde matrix:

$$\begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & 2 & \cdots & 2^n \\ \vdots & \vdots & \ddots & \vdots \\ 1 & n+1 & \cdots & (n+1)^n \end{pmatrix}.$$

Its determinant is $\prod_{1 \leq i < j \leq n+1} (j - i) \neq 0$. Thus, we can solve this linear system, and recover $Z_0, \cdots, Z_n$. In particular, Z_0 is what we want, namely the number of perfect matchings in G. □

This reduction is called the "interpolation" method. The reason is that if we view M_t as a polynomial in $(t+1)$ (often called the *matching polynomial*), then what we are doing is to recover the coefficients of M_t. Apparently this is not a many-one reduction, and to find a matching is a trivial problem in G (we can just take the empty one). Even finding the maximum weight matching is in P.

3 Variants of decision problems

Decision problems are important in the study computational complexity, but they are not the only kind of problems we study. Typical variants are search problems, optimization problems, and counting problems.

For example, SAT asks whether a given CNF formula φ is satisfiable. We may also want to find a satisfying assignment, if any. This is the search version. The optimization version, denoted MAXSAT, asks at most how many clauses can be satisfied, and the counting version, denoted #SAT, asks the number of satisfying assignments.

Obviously all three variants are harder than SAT itself, but we can indeed solve the optimization version by binary search if we have an algorithm to (the threshold version of) SAT. We can also solve the search version if we have an algorithm to SAT. This is captured by the notion of self-reducibility. On the other hand, #SAT, as we will see soon, is much harder than NP in some precise sense.

Now, suppose we have a subroutine to solve SAT, and the answer is yes for some φ. We can find a satisfying assignment by fixing variables one by one. When we fix x_1, we use the subroutine for two restrictions of φ where x_1 is given True / False respectively, and proceed with the one satisfiable. (It must exist, and if both are satisfiable, we pick an arbitrary one.)

This property that a problem can be solved given an algorithm solving smaller instances is called *self-reducibility*. We will not formally define it here, as multiple valid definitions exist. We have encountered this property in the proof of Karp-Lipton theorem.

On the other hand, search problem is not always as easy as its decision version. Testing whether a number is composite is now known to be in P [AKS04], but to find a prime factor is quite difficult, with no known efficient algorithms.

Remark (Bibliographic). Relevant chapters are [AB09, Chapter 17] and [Pap94, Chapter 18].

References

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach.* Cambridge University Press, 2009.

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Ann. of Math. (2)*, 160(2):781–793, 2004.

[Edm65] Jack Edmonds. Paths, trees, and flowers. *Canad. J. Math.*, 17(3):449–467, 1965.

[Lov79] László Lovász. On determinants, matchings, and random algorithms. In *FCT*, pages 565–574, 1979.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Sch80] Jacob T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.

[Tur37] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(1):230–265, 1937.

[Val79a] Leslie G. Valiant. The complexity of computing the permanent. *Theor. Comput. Sci.*, 8:189–201, 1979.

[Val79b] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979.

[Zip79] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *EUROSAM*, volume 72 of *LNCS*, pages 216–226. Springer, 1979.