Partition Functions, Polynomials and Optimization
Indian Institute of Science, Bangalore, Jan 13, 2017
Speaker: Nisheeth Vishnoi
Permanent and Counting Perfect Matchings in Bipartite Graphs

1 Counting Matchings

We will study the problem of counting perfect matchings in a bipartite graph. Before we proceed let us define it formally.

Definition 1.1. *Let $G = (V, E)$ be an undirected graph. A subset $M \subseteq E$ such that no two edges in M share an endpoint we call a matching. If M "matches" all vertices, i.e., $|M| = |V|/2$ then we say that M is a perfect matching.*

The problem: given a bipartite graph G, count the number of perfect matchings in G we denote by BMCOUNT.

Note that the decision version of the BMCOUNT problem, i.e., given a bipartite graph, decide whether it contains a perfect matching is polynomial time solvable (for example using any max-flow algorithm), in this regard it resembles the problem of counting spanning trees, where the decision version is just to say whether a graph is connected. However as it turns out, the counting variants of these two problems differ drastically. To explain it, we need to introduce the complexity class #**P**.

2 The Class #P

The complexity class #**P** can be seen as an analogue of **NP** for counting problems. Recall that a problem L is in **NP** if there exists an efficient (polynomial time) verifier for L. A verifier is simply a program V which given two inputs (x, y), where x should be thought of as an input for L and y as a certificate for x being an element of L, outputs YES or NO. It should hold that if $x \in L$ then there is a certificate y (of bounded length) which makes V output YES, and there should be no such certificate otherwise.

In the counting world we define #**P** to be the set of counting problems for which there exists a polynomial time verifier $V(x, y)$. In such a problem we are given x and we are asked to count the number of different y such that $V(x, y)$ says YES. As an instructive example, think of an input as a graph G and a certificate being some subset S of G's edges. We can then consider a verifier which takes (G, S) as input and answers YES if and only if S is a spanning tree in G. This clearly corresponds to the problem of counting spanning trees in a graph. Since such a verifier can be implemented in polynomial time, it means that this problem is in #**P**. Below we list several other examples of problems in this class.

1. Given a bipartite graph $G = (V, E)$ compute the number of perfect matchings in G.
2. Given a graph $G = (V, E)$ compute the number of Hamiltonian cycles in G.
3. Given a boolean formula ϕ compute the number of satisfying assignments.

If we consider decision versions of the above problems, i.e., "Does G ccontain a perfect matching?", "Does G contain a Hamiltonian cycle?" and "Does ϕ have a satisfying assignment?" all of them belong to **NP**, but of course the first of them is polynomial time solvable, while the remaining two are **NP**−complete. When thinking of the spanning tree problem, it would be plausible to conjecture that every counting problem, whose decision variant is polynomial time solvable is also polynomial time solvable. As it turns out, this intuition is completely wrong, as demonstrated by Valiant in his famous

Theorem 2.1 (Valiant '79)**.** *The problem* BMCOUNT *is* #**P**−*complete.*

We are not going to define #**P**−completeness precisely, but the meaning of it is easy to imagine. #**P**−complete problems are the hardest in the whole #**P** class, any other problem reduces to them. This theorem (which we are not going to prove) has a rather striking implication: if one could count perfect matchings in bipartite graphs in polynomial time, then **P** = **NP**. This demonstrates how hard BMCOUNT is compared to counting spanning trees.

3 Permanent of a Matrix

In this section we introduce another problem – computing permanents of nonnegative matrices. While this problem is algebraic and seemingly has nothing to do with counting perfect matchings, we show that these problems are tightly related to each other. In a later section we construct an algorithm for approximately computing permanents, which then automatically implies the same for BMCOUNT. The definition follows.

Definition 3.1. *Let $A \in \mathbb{R}^{n\times n}$ be a square matrix with entries $A_{i,j}$ for $1 \leq i,j \leq n$. The permanent of A is defined as:*

$$\mathrm{Per}(A) \stackrel{\text{def}}{=} \sum_{\sigma\in S_n} \prod_{i=1}^{n} A_{i,\sigma(i)},$$

where S_n is the set of all permutations over n symbols, i.e., the set of bijections $\sigma : \{1,2,\ldots,n\} \to \{1,2,\ldots,n\}$.

The computational problem: given a matrix A with nonnegative entries, compute $\mathrm{Per}(A)$ *is denoted as* PERM.

Note first that the number of different permuatations over n symbols is $n!$, hence we cannot compute permanents directly from the definition, because it would take exponential time.

Interestingly, the formula for computing permanents closely resembles another familiar object: the determinant. Indeed, recall that the determinant of a matrix $A \in \mathbb{R}^{n\times n}$ can be defined as

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

where $\mathrm{inv}(\sigma)$ denotes the number of inversions in σ and $(-1)^{\mathrm{inv}(\sigma)}$ is the sign of the permutation σ. Therefore Per and det differ only by the $(-1)^{\mathrm{inv}(\sigma)}$ term. Since determinants can be computed in polynomial time (using Gaussian elimination), one is tempted to believe that permanents should not be much harder. But again, this intuition turns out to be wrong, as we will see PERM is **#P**−complete, hence no polynomial algorithm is expected to exist for solving this problem.

Lemma 3.2. *Let G be an undirected bipartite graph on $2n$ vertices, then there exists a (polynomial time computable) matrix $A_G \in \{0,1\}^{n\times n}$ such that* $\mathrm{Per}(A_G)$ *is equal to the number of perfect matchings in G.*

Proof. Let V and U be the sides of the bipartition, we can assume that $|V| = |U| = n$, as otherwise there is no perfect matching in G. Let us denote the vertices of V and U by $\{v_1,\ldots,v_n\}$ and $\{u_1,\ldots,u_n\}$ respectively. Define the matrix $A_G = (a_{i,j})_{1\leq i,j\leq n}$ to be

$$a_{i,j} = \begin{cases} 1 & \text{if there is an edge between } u_i \text{ and } v_j \text{ in } G, \\ 0 & \text{otherwise.} \end{cases}$$

Note that every perfect matching M in G corresponds to a unique permutation $\sigma_M \in S_n$ such that $\sigma_M(i)$ is equal to k such that v_i is matched to u_j in M. Note that the corresponding term in $\mathrm{Per}(A_G)$ is

$$\prod_{i=1}^{n} a_{i,\sigma_M(i)} = 1.$$

Conversely, for every permutation $\sigma \in S_n$ such that $\prod_{i=1}^{n} a_{i,\sigma(i)} = 1$, there exists a unique perfect matching which corresponds to it. Since in $\mathrm{Per}(A_G)$ every term in the sum is either zero or one and the ones corresponds exactly to perfect matchings in G we can conclude the lemma.

□

In this talk we develop an approximation algorithm for computing permanents, which using the above Lemma, translates automatically to an algorithm for approximately counting perfect matchings in bipartite graphs. More precisely, we prove

Theorem 3.3. *There is a polynomial time algorithm which given an $n \times n$ matrix A with nonnegative entries, outputs a number $X \in \mathbb{R}$ such that:*

$$\mathrm{Per}(A) \le X \le \frac{n^n}{n!}\mathrm{Per}(A).$$

4 Permanents and Polynomials

Similarly as for the case of spanning trees, we will use polynomials as a tool for computing permanents. To every square matrix A we assign a polynomial p_A. It will play a crucial role in obtaining algorithms for approximately computing permanents.

Definition 4.1. *Let $A \in \mathbb{R}^{n\times n}$ be a matrix with non-negative entries. The product polynomial $p_A \in \mathbb{R}[x_1, x_2, \ldots, x_n]$ of A is defined as*

$$p_A(x_1, \ldots, x_n) = \prod_{i=1}^{n} \left(\sum_{j=1}^{n} x_j a_{i,j} \right).$$

It is not hard to see that p_A has all monomials of degree n (such polynomials we call $n-$homogenous) and all its coefficients are non-negative (because the entries of A are non-negative). Importantly, the permanent of A can be recovered from A simply as one of its coefficients! More precisely we have

Fact 4.2. *Let $A \in \mathbb{R}^{n\times n}$, we have:*

$$\mathrm{Per}(A) = \frac{\partial^n}{\partial x_1 \partial x_2 \ldots \partial x_n} p_A(x).$$

Proof. Since p_A has all monomials of degree n it is not hard to see that the expression $\frac{\partial^n}{\partial x_1 \partial x_2 \ldots \partial x_n} p_A(x)$ just gives us the coefficient of the monomial $\prod_{i=1}^n x_i$ in p_A, we will denote it by $(p_A)_{[n]}$. It remains to show that $(p_A)_{[n]}$ indeed corresponds to the permanent of A.

Imagine completely expanding the expression $p_A(x) = \prod_{i=1}^n \left(\sum_{j=1}^n x_j a_{i,j}\right)$ and thus getting n^n terms each of the form:

$$\prod_{i=1}^m x_{\beta(i)} a_{i,\beta(i)},$$

where β is any mapping $\{1,2,\ldots,n\} \to \{1,2,\ldots,n\}$. The monomial $\prod_{i=1}^n x_i$ in p_A is simply the sum of all terms as above for β being permutations. Hence the coefficient is:

$$(p_A)_{[n]} = \sum_{\beta \in S_n} \prod_{i=1}^m a_{i,\beta(i)} = \mathrm{Per}(A).$$

□

The above fact has a rather surprising implication. We have described the permanent as a coefficient of an "easy" polynomial. By easy we mean that $p_A(x)$ can be evaluated efficiently for any input x, indeed this requires only n^2 arithmetic operations. However, this does not help us in computing the required coefficient. In fact, since p_A has an exponential number of different monomials it is hard to imagine how to gain some knowledge about a specific one without computing all of them (and spending exponential time to do that).

As it turns out, we can actually learn something about the relevant coefficient by solving a continuous *optimization problem* over the polynomial $p_A(x)$. More precisely, let us define the following notion of capacity of a polynomial.

Definition 4.3. *Let $p \in \mathbb{R}[x_1,\ldots,x_n]$ be any $n-$variate real polynomial. We define its capacity to be*

$$\mathrm{Cap}(p) = \inf_{x_1,\ldots,x_n > 0} \frac{p(x)}{\prod_{i=1}^n x_i}.$$

The following theorem makes use of certain good analytic properties of the polynomial p_A to establish a surprising connection between capacity of p_A and the coefficient $(p_A)_{[n]}$. We state it here for the special case of p_A, in the next talk we will state and prove it in much more generality: for real stable polynomials.

Theorem 4.4 (Gurvits '08). *Let A be an $n \times n$ matrix with non-negative entries. Let p_A be the corresponding product polynomial and $(p_A)_{[n]}$ be the coefficient of $\prod_{i=1}^n x_i$. It holds*

$$(p_A)_{[n]} \leq \mathrm{Cap}(p_A) \leq \frac{n^n}{n!}(p_A)_{[n]}.$$

As mentioned above, the full proof will appear in the next talk, let us now only establish the left hand side of the inequality, as it is very simple. Since $p_A(x)$ has all coefficients nonnegative, it is easy to see that for any positive $x_1, \ldots, x_n \in \mathbb{R}$ we have $(p_A)_{[n]} \leq \frac{p_A(x)}{\prod_{i=1} x_i}$. After taking the infimum over $x > 0$ we obtain $(p_A)_{[n]} \leq \mathrm{Cap}(p_A)$.

Note that Fact 4.2 together with Theorem 4.4 imply that the capacity of the product polynomial p_A is an $\frac{n^n}{n!}-$approximation to the permanent of A. Hence in order to obtain an approximation algorithm for computing permanents we are left with the task of computing the capacity of the product polynomial.

5 Computing Capacity

In this section we prove

Lemma 5.1. *The problem of computing* $\mathrm{Cap}(p_A)$ *for an $n \times n$ matrix A with nonnegative entries can be stated as a convex program and can be solved in polynomial time using convex optimization tools.*

Let us denote $p = p_A$. The first step towards proving the above lemma is to reformulate the problem:

$$\inf_{x_1,\ldots,x_n>0} \frac{p(x)}{\prod_{i=1}^n x_i}$$

so that there are no constraints on the variables. To this end to observe that:

$$\{x \in \mathbb{R}^n : x_1, x_2, \ldots, x_n > 0\} = \{(e^{y_1}, \ldots, e^{y_n}) : y \in \mathbb{R}^n\}.$$

Hence we can replace x by $e^y = (e^{y_1}, \ldots, e^{y_n})$. We obtain:

$$\inf_{y \in \mathbb{R}^n} p(e^y) \cdot e^{-\sum_{i=1}^n y_i}.$$

Note that the above is an unrestricted continuous optimization problem, below we prove that is is convex, and actually much more: that the logarith of the above objective is convex (this implies convexity of the orginal function).

Fact 5.2. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function of the form*

$$f(y) = \log\left(\sum_{k=1}^{d} a_k e^{\langle v_k, y\rangle}\right)$$

where $a_k \in \mathbb{R}_{\geq 0}$ and $v_k \in \mathbb{R}^n_{\geq 0}$. Then f is convex.

Proof. Since composing a convex function with an affine function results in a convex function, it is enough to prove that the function $g : \mathbb{R}^d \to \mathbb{R}$ defined as

$$g(z) = \log\left(\sum_{k=1}^{d} e^{z_k}\right)$$

is convex. We will directly use the definition of convexity. Take any $\lambda \in (0,1)$, the goal is to show that $g(\lambda z + (1-\lambda)w) \leq \lambda g(z) + (1-\lambda)g(w)$ for any $z, w \in \mathbb{R}^d$. We apply Hölder's inequality with $p = \frac{1}{\lambda}$ and $q = \frac{1}{1-\lambda}$:

$$\sum_{k=1}^{d} e^{\lambda z_k + (1-\lambda)w_k} = \sum_{k=1}^{d} e^{\lambda z_k} \cdot e^{(1-\lambda)w_k} \leq \left(\sum_{k=1}^{d} e^{z_k}\right)^{\lambda} \left(\sum_{k=1}^{d} e^{w_k}\right)^{1-\lambda}$$

Taking logarithms:

$$\ln \sum_{k=1}^{d} e^{\lambda z_k + (1-\lambda)w_k} \leq \lambda \ln \sum_{k=1}^{d} e^{z_k} + (1-\lambda) \ln \sum_{k=1}^{d} e^{w_k}.$$

□

We are now ready to deduce Lemma 5.1.

Proof of Lemma 5.1. As in the above discussion we reformulate the problem of computing capacity to the following:

$$\log \mathrm{Cap}(p) = \inf_{y \in \mathbb{R}^n} \log p(e^y) - \sum_{i=1}^{n} y_i.$$

By Fact 5.2 the objective $f(y) = \log p(e^y) - \sum_{i=1}^{n} y_i$ is a convex function. Moreover, as proved in Homework 1, this function is also $L-$smooth for $L = O(n^2)$.

This suggests that we should apply gradient descent for $L-$smooth functions (see Lecture 1) to find an approximate minimum of $f(y)$.

The only question which remains is: how to bound the initial distance from a starting point, say 0, to the optimal point $y^\star$? The first obstacle which we encounter is that actually such an optimal point $y^\star$ may not exist!Indeed, there are polynomials p such that for every $u \in \mathbb{R}^n$ we have $f(u) > \inf_{y \in \mathbb{R}_n} f(y)$.[1] Of course we are not aiming for the exact minimum anyway (only for some approximation), but still, dealing with this issue requires some effort.

There are several ways to fix this problem. One possibility is to enforce that the optimal value is actually attained at some point $y^\star$. This can be guaranteed by assuming that A is strictly positive, which in turn can be achieved by adding a small value $\varepsilon > 0$ to all entries of the matrix. Note that such a perturbation does not change the permanent of A by too much.

$\square$

[1]For example, consider the polynomial $p(x_1, x_2, x_3) = x_1 x_2 x_3 + x_1^2 x_2$.