A Tutorial in Barvinok's Method

Yanheng Wang

1 Barvinok's Method for Approximating Polynomials

Let $p(z) : \mathbb{C} \to \mathbb{C}$ be a polynomial of degree n. We would like to compute $p(1)$ approximately. A classical tool from analysis is Taylor expansion, but apparently there's no point to "approximate" a polynomial by its Taylor expansion (which is a polynomial again). However, Barvinok observed that we could expand $\ell(z) := \ln p(z)$ instead of $p(z)$. The logarithm intuitively diffuses the coefficients of the original polynomial. Once we obtain a good approximation to $\ell(1)$, raising it to the exponent would give us a good approximation to $p(1)$. Therefore, in the following we concentrate in approximating $\ell(1)$.

1.1 A Taylor Expansion Approximation

Suppose the complex roots of $p(z)$ are $r_1, r_2, \ldots, r_n$. Then we may write

$$p(z) = a_0 \prod_{i=1}^{n} (z - r_i)$$

and

$$\ell(z) = \ln a_0 + \sum_{i=1}^{n} \ln(z - r_i).$$

Differentiating k times and taking $z = 0$ gives

$$\ell^{(k)}(0) = -(k-1)! \sum_{i=1}^{n} r_i^{-k}. \tag{1}$$

Therefore, the Taylor series of $\ell(z)$ at $z = 0$ writes

$$\begin{aligned} \ell(z) &= \ln a_0 + \sum_{k=1}^{\infty} \frac{\ell^{(k)}(0)}{k!} z^k \\ &= \ln a_0 - \sum_{k=1}^{\infty} \sum_{i=1}^{n} \frac{1}{k r_i^k} z^k. \end{aligned}$$

Now we truncate the series by discarding $k > m$:

$$\hat{\ell}(z) := \ln a_0 - \sum_{k=1}^{m} \sum_{i=1}^{n} \frac{1}{k r_i^k} z^k.$$

Then the error between $\hat{\ell}(1)$ and $\ell(1)$ is bounded by

$$|\ell(1) - \hat{\ell}(1)| \leq \sum_{k=m+1}^{\infty} \sum_{i=1}^{n} \frac{1}{|k| \cdot |r_i|^k} \leq \frac{1}{m+1} \sum_{k=m+1}^{\infty} \sum_{i=1}^{n} \frac{1}{|r_i|^k}.$$

It comes clear that the error is small provided the roots r_i all stay away from the unit disc on complex plane. More specifically, suppose $|r_1|, |r_2|, \ldots, |r_n| \geq \beta > 1$, then

$$|\ell(1) - \hat{\ell}(1)| \leq \frac{n}{m+1} \sum_{k=m+1}^{\infty} \frac{1}{\beta^k} = \frac{n}{(m+1)(\beta-1)\beta^m}$$

and thus $\hat{\ell}(1)$ is a good approximation to $\ell(1)$. We summarise the discussion into the following lemma.

Lemma 1. Suppose all roots of a complex polynomial $p(z)$ stay away from the disc $\{z \in \mathbb{C} : |z| \leq \beta\}$ where $\beta > 1$. Denote $\ell(z) := \ln p(z)$ and take $\hat{\ell}(z)$ to be its truncated Taylor expansion at 0 of order m. Then $|\ell(1) - \hat{\ell}(1)| \leq \epsilon$ when we choose $m = O(\log(n/\epsilon))$.

1.2 A Quasi-Polynomial Time Algorithm

To use the previous lemma algorithmically, we must show that $\hat{l}(1)$ can be computed easily. Note that we do not assume knowledge about $r_1, r_2, \ldots, r_n$.

Write $p(z) = \sum_{i=0}^{n} a_i z^i$. Since by definition $\ell(z) = \ln p(z)$, we have $\ell'(z) = p'(z)/p(z)$ or equivalently $p'(z) = p(z)\ell'(z)$. Differentiating $k-1$ times,

$$p^{(k)}(z) = \sum_{i=0}^{k-1} \binom{k-1}{i} p^{(i)}(z) \ell^{(k-i)}(z).$$

In paticular, taking $z = 0$ we have

$$k!\ a_k = \sum_{i=0}^{k-1} \binom{k-1}{i} i!\ a_i\ \ell^{(k-i)}(0). \tag{2}$$

If we know the values $a_0, a_1, \ldots, a_m$, then this is a triangular linear system, from which we could solve $\ell^{(k)}(0)$ for $k = 1, 2, \ldots, m$. The computation of $\hat{\ell}(1)$ then follows.

Since we took $m = O(\log(n/\epsilon))$ in Lemma 1, we may use a brute-force approach to compute $a_0, a_1, \ldots, a_m$. For instance, let $p(z)$ be the independence polynomial, i.e. a_k being the count of independent sets of size k in graph G. To compute a_k, we may enumerate all k-subsets of $V(G)$ and count the number of independent ones. This yields an $n^{O(\log(n/\epsilon))}$ (i.e. quasi-polynomial time) algorithm.

2 Proving Zero-Free Property

This section develops basic tools to prove the zero-free property required by Lemma 1. It can be skipped safely.

Definition 1. Let $p(\mathbf{z}) : \mathbb{C}^n \to \mathbb{C}$ be a multivariate polynomial and $\mathbb{D} := \{z \in \mathbb{C} : |z| \leq 1\}$ be the unit disc on complex plane. We say $p(\mathbf{z})$ is *stable* if $p(\mathbf{z}) \neq 0$ for all $\mathbf{z} \in \mathbb{D}^n$.

Definition 2. Write $\mathbf{z}^S := \prod_{i \in S} z_i$. Let $p(\mathbf{z}) := \sum_{S \subseteq [n]} a_S \mathbf{z}^S$ and $q(\mathbf{z}) := \sum_{S \subseteq [n]} b_S \mathbf{z}^S$ be two n-variate, multi-affine polynomials. The Schur product $p * q$ is defined by

$$p * q := \sum_{S \subseteq [n]} a_S b_S \mathbf{z}^S.$$

Lemma 2. If the bivariate polynomial $r(x, y) := axy + bx + cy + d$ is stable, then its *contraction* $r^-(z) := az + d$ is also stable.

Proof. If $a = 0$ then $d \neq 0$, and $r^-(z) = d$ is stable. If $a \neq 0$, the only root of $r^-(z)$ is $-d/a$, so it suffices to prove $|d| > |a|$. By symmetry we assume $|c| \geq |b|$. Since $r(x, y)$ is stable, we have $|bx + d| > |ax + c|$ for all $x \in \mathbb{D}$. Let's take $x_0 : |x_0| = 1, |ax_0 + c| = |a| + |c|$. Then $|b| + |d| \geq |bx_0 + d| > |a| + |c|$ and thus $|d| - |a| > |c| - |b| \geq 0$. □

Lemma 3. Let p and q be two n-variate, multi-affine polynomials. If p and q are stable, then both pq and $p * q$ are stable.

Proof. It's trivial that pq is stable. Now we prove that $p * q$ is stable by induction on n. If $n = 1$, then $p * q = a_\emptyset b_\emptyset + a_1 b_1 z_1$ whose root ρ satisfies $|\rho| = |a_\emptyset|/|a_1| \cdot |b_\emptyset|/|b_1| > 1$ since p and q are stable. Next we prove the case $n + 1$, assuming the lemma holds for $\leq n$. We may write

$$p(\mathbf{z}) = \sum_{S \subseteq [n-1]} \mathbf{z}^S \left(z_n a_{S \cup \{n\}} + a_S \right), \qquad q(\mathbf{z}) = \sum_{S \subseteq [n-1]} \mathbf{z}^S \left(z_n b_{S \cup \{n\}} + b_S \right).$$

We define $p(\mathbf{z} \mid z_n = x)$ to be $p(\mathbf{z})$ with z_n fixed to a constant $x \in \mathbb{D}$. Similarly define $q(\mathbf{z} \mid z_n = y)$. By induction hypothesis, their Schur product

$$\sum_{S \subseteq [n-1]} \mathbf{z}^S \left(a_{S \cup \{n\}} b_{S \cup \{n\}} \cdot xy + a_{S \cup \{n\}} b_S \cdot x + a_S b_{S \cup \{n\}} \cdot y + a_S b_S \right)$$

is stable. Since $x, y \in \mathbb{D}$ are chosen arbitrarily, we could raise them back to variables. At the same time, we fix the remaining variables $z_1, \ldots, z_{n-1}$ to constants. So the above quantity can be viewed as a stable bivariate polynomial $r(x, y) = axy + bx + cy + d$ where the constants a, b, c, d are easy to write but too long to place. By Lemma 2, its contraction $r^-(z) := az + d$ is stable as well. But $r^-(z_n)$ is exactly

$$\sum_{S \subseteq [n-1]} \mathbf{z}^S \left(a_{S \cup \{n\}} b_{S \cup \{n\}} \cdot z_n + a_S b_S \right) = \sum_{S \subseteq [n]} a_S b_S \mathbf{z}^S = p * q$$

which completes the induction. □

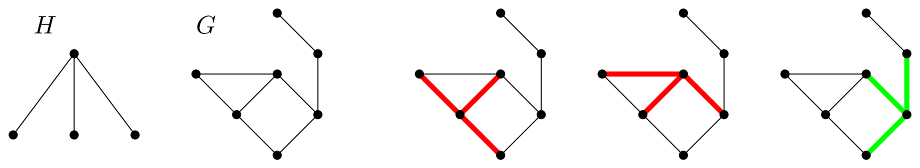
3 Faster Algorithm via Induced Subgraph Counts

In this section we introduce a framework by Patel and Regts that speeds up the computation of the quasi-polynomial time algorithm.

3.1 Preliminaries

We write $\mathcal{G}$ for the class of all graphs, and $\mathcal{G}_s$ for the class of graphs with at most s vertices. We denote by $\mathcal{C}$ the class of all connected graphs. We will assume throughout that the maximum degree, Δ, of graph G is a bounded constant.

If $H, G \in \mathcal{G}$, we denote by $\operatorname{ind}(H,G) := |\{S \subseteq V(G) : G[S] \cong H\}|$, that is the number of *induced* subgraphs of G that are isomorphic to H. The following figure gives an example. H appears three times in G as a subgraph, but only the green appearance counts, so $\operatorname{ind}(H,G) = 1$ in this example.



We call a graph invariant $f : \mathcal{G} \to \mathbb{C}$ multiplicative if $f(\emptyset) = 1$ and $f(G \dot\cup G') = f(G)f(G')$. Similarly, we call it additive if $f(G \dot\cup G') = f(G) + f(G')$.

We next define a broad class of graph polynomials that will fit into our framework.

Definition 3. The class BIGCP (bounded induced graph counting polynomial) consists of all multiplicative graph polynomials $p(z) = \sum_{i=0}^{n} a_i z^i$ such that

(1) $a_0 = 1$ and the other coefficients a_i could be written in the form

$$a_i = \sum_{H \in \mathcal{G}_{\alpha i}} \lambda_i(H) \cdot \operatorname{ind}(H,G) \tag{3}$$

where $\alpha \in \mathbb{N}$ and $\lambda_i(H) \in \mathbb{R}$ are constants independent of G;

(2) $\lambda_i(H)$ can be computed in polynomial time of $|H|$.

For instance, the independence polynomial is a BIGCP since $a_i = \operatorname{ind}(\overline{K_i}, G)$; we may take $\alpha := 1$ and $\lambda_i(H) := \mathbf{1}[H \cong \overline{K_i}]$.

The definitions actually bear a simple intuition. Many graph-theoretic objects of interest are vertex/edge subsets that satisfy certain constraints. For instance, an independent set is a *vertex subset* in which no vertices are adjacent; a matching is an *edge subset* in which no edges are incident. To treat them uniformly we may encode them by induced subgraphs. For example, a matching M is encoded by an induced subgraph $(V(M), M)$, which is merely a wrapping of the object. (We use *induced* subgraphs rather than arbitrary subgraphs because they encode *complete* information among the region they span; otherwise it's ambiguous to decode a subgraph.) Then the associated graph polynomial writes $p(z) := \sum_{i=0}^{n} a_i z^i$ where

$$a_i := \sum_{\substack{H \subseteq G \\ \text{induced}}} \lambda_i(H).$$

But typically $\lambda(H) = \lambda(H')$ provided $H \cong H'$, i.e. the location of the object doesn't matter. So we could aggregate all isomorphic H's via $\operatorname{ind}(H,G)$, which gives us the form in Equation (3). It essentially extracts the structure H and neglects its location.

3.2 Properties of $\mathrm{ind}(H, G)$

The following property is very useful since it boils a product down to linear combination. As we will see later, repeated application of the lemma kills any higher-order terms that are difficult to manipulate.

Lemma 4. Suppose $H_1 \in \mathcal{G}_r$, $H_2 \in \mathcal{G}_s$ (possibly sharing vertices and edges). Then

$$\mathrm{ind}(H_1, G) \cdot \mathrm{ind}(H_2, G) = \sum_{H \in \mathcal{G}_{r+s}} c(H_1, H_2; H) \cdot \mathrm{ind}(H, G)$$

where $c(H_1, H_2; H) := |\{(S_1, S_2) : S_1 \cup S_2 = V(H), H[S_1] \cong H_1, H[S_2] \cong H_2\}|$.

Proof. The LHS counts the size of the set $\{(S_1, S_2) : G[S_1] \cong H_1, G[S_2] \cong H_2\}$. We could imagine moving two templates, H_1 and H_2, around the graph G and count every time when both of them found a match.

The RHS does the same thing in two stages: (i) it enumerates H as a candidate structure for $G[S_1] \cup G[S_2]$; (ii) it decomposes H into S_1, S_2 and count. The $c(H_1, H_2; H)$ takes care of possible decompositions, while the $\mathrm{ind}(H, G)$ accounts for possible locations of the bulk $G[S_1] \cup G[S_2]$. □

We next show another crucial property of induced subgraph counts. It's the source that we gain speedup over the original algorithm: a brute-force enumeration of *connected* graphs is much more efficient, as such graphs "grows" locally.

Lemma 5. Let $f(G) := \sum_{H \in \mathcal{G}} \lambda(H) \cdot \mathrm{ind}(H, G)$ be a graph invariant. Then $f(G)$ is additive if and only if $\lambda(H) = 0$ for all disconnected H.

Proof. ($\Leftarrow$) For disjoint graphs G, G' and connected graph H, we have $\mathrm{ind}(H, G \dot\cup G') = \mathrm{ind}(H, G) + \mathrm{ind}(H, G')$ since H cannot span two disjoint parts. Hence $f(G \dot\cup G') = f(G) + f(G')$.

($\Rightarrow$) We assume without loss of generality that $\lambda(H) = 0$ for all $H \in \mathcal{C}$. (If this is not the case, we subtract from $f(G)$ another additive invariant $\delta(G) := \sum_{H \in \mathcal{C}} \lambda(H) \cdot \mathrm{ind}(H, G)$.) Now we proceed by induction on the size of H. Suppose $\lambda(H) = 0$ for all $H : |H| \leq s$. Consider a disconnected graph $H : |H| = s + 1$. We partition it into two smaller disconnected parts, say H_1 and H_2. By additivity, $f(H) = f(H_1) + f(H_2) = 0 + 0 = 0$ since $\mathrm{ind}(H, H_1) = \mathrm{ind}(H, H_2) = 0$. On the other hand, $f(H) = \lambda(H)\mathrm{ind}(H, H) = \lambda(H)$. Hence $\lambda(H) = 0$. □

The final lemma would show the concrete advantage of a connected graph.

Lemma 6. If H is a connected graph of size s, then we may enumerate (possibly with repetitions) all $S \subseteq V(G) : G[S] \cong H$ in $O(n\Delta^s)$ time.

Proof. Let $v_1, v_2, \ldots, v_s$ be a breadth-first order of H. We try to enumerate corresponding vertices $w_1, w_2, \ldots, w_s$ from G to make $v_i \mapsto w_i$ an isomorphism of $H \to G$. Initially we choose a candidate $w_1 \in G$; there are n choices. Now suppose we have selected $w_1, \ldots, w_i$ and are ready to pick w_{i+1}. Let $j \leq i : v_j \in N_H(v_{i+1})$; by connectivity of H and the definition of breadth-first order, such j must exist. We then propose w_{i+1} to be a neighbour of w_j; there are $\leq \Delta$ choices. When we finish filling up w_s, we print the sequence if $v_i \mapsto w_i$ is indeed an isomorphism. The overall time complexity is $O(n\Delta^s)$. □

Corollary 7. If H is a connected graph of size s, then we may compute $\operatorname{ind}(H,G)$ in $O((n\Delta^s)^2)$ time. In particular, we could determine if $H \cong H'$ (i.e. if $|H| = |H'| \wedge \operatorname{ind}(H,H') > 0$) in $O(n\Delta^s)$.

3.3 Approximating BIGCP by Dynamic Programming

Let $p(z) := \sum_{i=0}^{n} a_i z^i$ be a BIGCP with roots $r_1, r_2, \ldots, r_n$. We define the *inverse power sum* $\sigma_k := \sum_{i=1}^{n} r_i^{-k}$. Since $p(z)$ is multiplicative, we see that σ_k is additive for all $k \in \mathbb{N}$.

Remark. In fact, we could proceed our following exposition without introducing $\{\sigma_k\}_{k=1}^{n}$, since $\ell^{(k)}(0)$ is a constant multiple of σ_k. The latter merely lightens our notation.

We start by relating $\{a_i\}_{i=0}^{n}$ and $\{\sigma_k\}_{k=1}^{n}$ (often called *Newton Formulas*). Plug Equation (1) into Equation (2) and simplify, we would get a recursive formula

$$\sigma_k = -ka_k - \sum_{i=1}^{k-1} a_i \sigma_{k-i}. \tag{4}$$

At first glance, the recursion is non-linear due to the product terms $a_i\sigma_{k-i}$. But actually we could spread the product with the help of Lemma 4. It's easy to prove by induction on k that

$$\sigma_k = \sum_{H \in \mathcal{G}_{\alpha k}} \gamma_k(H) \cdot \operatorname{ind}(H,G).$$

for *some* constants $\gamma_k(H)$ independent of G. Then by Lemma 5, $\gamma_k(H) = 0$ whenever H is disconnected. Therefore, the above equation simplifies to

$$\sigma_k = \sum_{H \in \mathcal{C}_{\alpha k}} \gamma_k(H) \cdot \operatorname{ind}(H,G). \tag{5}$$

where $\mathcal{C}_{\alpha k}$ denotes the class of connected induced subgraphs *in* G of size at most αk.

In the rest of this section, we show an algorithm that computes $\gamma_k(H)$ via dynamic programming. Once we have them, we could compute σ_k and $\ell^{(k)}(0) = -(k-1)!\sigma_k$ accordingly.

Remark. It seems intriguing to avoid γ's altogether and compute σ_k directly:

$$\begin{aligned}
\sigma_k &= -k \sum_{H \in \mathcal{G}_{\alpha k}} \lambda(H,k)\operatorname{ind}(H,G) - \sum_{i=1}^{k-1} \sum_{H \in \mathcal{G}_{\alpha i}} \lambda(H,i)\operatorname{ind}(H,G) \cdot \sigma_{k-i} \\
&= -k \sum_{H \in \mathcal{C}_{\alpha k}} \lambda(H,k)\operatorname{ind}(H,G) - \sum_{i=1}^{k-1} \sum_{H \in \mathcal{C}_{\alpha i}} \lambda(H,i)\operatorname{ind}(H,G) \cdot \sigma_{k-i}.
\end{aligned}$$

But the second line is problematic since σ_{k-i} depends on the graph G and should not be treated as a constant. So we could *not* apply Lemma 5 to simplify the summation to $\mathcal{C}_{\alpha k}$. On the other hand, the first line is correct but inefficient: there are about $2^{(\alpha m)^2} = n^{\Theta(\log n)}$ possible $H \in \mathcal{G}_{\alpha k}$, which does not beat our old algorithm.

We now show how to compute $\gamma_k(H)$. Plugging the "explicit formulas" (3)(5) into RHS of the recursion (4), we get

$$\begin{aligned}\sigma_k &= -k \sum_{H\in\mathcal{G}_{\alpha k}} \lambda_k(H)\mathrm{ind}(H,G)\\ &\quad - \sum_{i=1}^{k-1} \sum_{A\in\mathcal{G}_{\alpha i}} \sum_{B\in\mathcal{C}_{\alpha(k-i)}} \lambda_i(A)\gamma_{k-i}(B)\ \mathrm{ind}(A,G)\mathrm{ind}(B,G)\\ &= -\sum_{H\in\mathcal{G}_{\alpha k}} \left(k\lambda_k(H) + \sum_{i=1}^{k-1} \sum_{A\in\mathcal{G}_{\alpha i}} \sum_{B\in\mathcal{C}_{\alpha(k-i)}} \lambda_i(A)\ \gamma_{k-i}(B)\ c(A,B;H)\right) \mathrm{ind}(H,G)\end{aligned}$$

where the last line follows from Lemma 4 and exchanging the summations. Compariing it with Equation (5), for any $k \in [m]$ and $H \in \mathcal{C}_{\alpha k}$,

$$\gamma_k(H) = k\lambda_k(H) + \sum_{i=1}^{k-1} \sum_{A\in\mathcal{G}_{\alpha i}} \sum_{B\in\mathcal{C}_{\alpha(k-i)}} \lambda_i(A)\ \gamma_{k-i}(B)\ c(A,B;H).$$

This equation naturally suggests a dynamic programming algorithm that works from $k = 1$ to $k = m$ and keeps the entries in a table. It remains to show that the summation can be done fast. In fact, the inner summation is mostly redundant. By definition,

$$c(A,B;H) = \sum_{S\cup T=V(H)} \mathbf{1}\{H[S]\cong A\}\cdot\mathbf{1}\{H[T]\cong B\}.$$

So we could rewrite the equation as

$$\gamma_k(H) = k\lambda_k(H) + \sum_{i=1}^{k-1} \sum_{\substack{S\cup T=V(H)\\ H[S]\in\mathcal{G}_{\alpha i}\\ H[T]\in\mathcal{C}_{\alpha(k-i)}}} \lambda_i(H[S])\ \gamma_{k-i}(H[T]) \tag{6}$$

which saves many redundant enumerations of A and B.

Now we note that $|H| \le \alpha k \le \alpha m = O(\log(n/\epsilon))$. Since our inner summation requires $S\cup T = V(H)$, we have at most three possibilities for each $v \in V(H)$: $v \in S\setminus T$; $v \in T\setminus S$; or $v \in S \cap T$. So we have $3^{O(\log(n/\epsilon))} = n^{O(1)}$ possibilities for S, T in total.

After finishing all $\gamma_k(H)$, we use Equation (5) to compute the σ_k's. There are $O(\Delta^{\alpha k})$ many $H \in \mathcal{C}_{\alpha k}$ and we could enumerate them in polynomial time. (On the contrary, the number would explode if we drop the connectivity contraint.)

There is a technical detail left: the dynamic programming table is indexed by k and H. We need an efficient way to look for the entry $\gamma_{k-i}(H[T])$. Luckily, the summation is over all $H[T] \in \mathcal{C}_{\alpha(k-i)}$, so we could simply search over polynomially-many graphs and use Corollary 7 to test for isomorphism.

3.4 Generalisation

The framework can be extended naturally to other contexts where an induced subgraph does not capture the full structure. Refer to Patel and Regts' paper and the work by Liu, Sinclair and Srivastava for discussion.