The Finite-Field FFT

Lecture 10

Intro

Given a univariate polynomial which we wish to evaluate at a point x, there are many ways to rearrange the calculation. For example, we can re-write

$$A(x) = \sum_{i=0}^{n-1} a_i x^i$$

in the form

$$A(x) = (x^0 x^1 ... x^{n-1}) \cdot \begin{pmatrix} a_0 \\ a_1 \\ ... \\ a_{n-1} \end{pmatrix}$$

Using Horner's rule, we can rewrite this as

$$A(x) = a_0 + x(a_1 + x(a_2 + ... + x a_{n-1} ...)).$$

If we knew {a_i} ahead of time we could pre-process the coefficients

a monic polynomial p of degree $n = 2^k - 1$ can be re-expressed as

$$p(x) = (x^{(n+1)/2} + a)q(x) + r(x)$$

where a is a constant and $q(x)$ and $r(x)$ are monic polynomials of degree $2^{k-1} - 1$. Recursively rearranged in this way, evaluation takes about $(n-3)/2+\log_2(n)$ operations, but these are not integer operations any more!

Simultaneous Evaluation at Many Points

If we keep the polynomial fixed, and evaluate at many points, we can do better, as we will see. For specificity in notation, let us evaluate A(x) at n given points, x_0 , ... , x_{n-1} .

Let us represent the problem as a matrix multiplication:

$$\begin{pmatrix} A(x_0) \\ A(x_1) \\ \dots \\ A(x_{n-1}) \end{pmatrix} = \begin{pmatrix} x_0^0 & x_0^1 & \dots & x_0^{n-1} \\ x_1^0 & x_1^1 & \dots & x_1^{n-1} \\ \dots & \dots & \dots & \dots \\ x_{n-1}^0 & x_{n-1}^1 & \dots & x_{n-1}^{n-1} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \dots \\ a_{n-1} \end{pmatrix}$$

Is this perfectly clear?

Roots of Unity

It is especially convenient for subsequent operations to choose the points specially:

Definition:

ω_n is a primitive n^{th} root of unity in a computation structure (e.g. finite field), if $\omega_n^{\,n} = 1$ but for no m such that $0 < m < n$ is $\omega_n^{\,m} = 1$.

Examples:

In the finite field of the integers mod 41 ($\{ \mathbf{Z} \}_{41}$), the element 3 is a primitive 8^{th} root of unity.

In $\mathbf{Z}_3$ the element 2 is a square-root of unity, since $2^2 = 4 \equiv 1 \mod 3$.

Over the complex numbers, $\omega_n = e^{2 \pi i / n}$.

Definition of Fourier Transform

Definition:
A Fourier Transform (FT) is a sequence of n points constituting the evaluation of a polynomial with coefficients $\{ a_0 , ..., a_{n-1} \}$ at the points $\{ \omega^0 , ..., \omega^{n-1} \}$ where ω is a primitive n^{th} root of unity.

That is, the Fourier Transform can be thought of as the left hand side of the equation on the next slide.

FT is "just" a matrix multiplication

$$\begin{pmatrix} A(1) \\ A(\omega) \\ \dots \\ A(\omega^{n-1}) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)^2} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \dots \\ a_{n-1} \end{pmatrix}$$

for convenience we will denote it this way

$$\hat{\mathbf{A}} = \mathbf{F}_n \cdot \mathbf{A}$$

stop here and understand this.

Computing $\hat{\mathbf{A}} = \mathbf{F}_n \cdot \mathbf{A}$

Let n be even = 2k.

$$A_{\text{even}}(x) = \sum_{i=0}^{k-1} a_{2i} x^i$$

$$A_{\text{odd}}(x) = \sum_{i=0}^{k-1} a_{2i+1} x^i$$

Then we can see that

$$A(x) = A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2).$$

Computing $\hat{\mathbf{A}} = \mathbf{F}_n \cdot \mathbf{A}$

$$\hat{\mathbf{A}}_n = \mathbf{F}_{2k} \cdot \mathbf{A}_{2k} = \begin{pmatrix} A(1) \\ A(\omega) \\ \cdots \\ A(\omega^{k-1}) \\ A(\omega^k) \\ A(\omega^{k+1}) \\ \cdots \\ \\ A(\omega^{2k-1}) \end{pmatrix} = \begin{pmatrix} A_{\text{even}}(1) + A_{\text{odd}}(1) \\ A_{\text{even}}(\omega^2) + \omega A_{\text{odd}}(\omega^2) \\ \cdots \\ A_{\text{even}}((\omega^{k-1})^2) + \omega^{k-1} A_{\text{odd}}((\omega^{k-1})^2) \\ A_{\text{even}}(1) + \omega^k A_{\text{odd}}(1) \\ A_{\text{even}}(\omega^2) + \omega^{k+1} A_{\text{odd}}(\omega^2) \\ \cdots \\ A_{\text{even}}((\omega^2)^{k-1}) + \omega^{2k-1} A_{\text{odd}}((\omega^2)^{k-1}) \end{pmatrix}$$

Note that we have, in the last three displayed rows, made use of the facts that $(\omega^2)^k = \omega^{2k} = \omega^n = 1$, $(\omega^{k+1})^2 = \omega^{2k+2} = \omega^2$, and $(\omega^{2k-1})^2 = \omega^{4k-2} = \omega^{2k-2} = (\omega^2)^{k-1}$.

Computing $\hat{\mathbf{A}} = \mathbf{F}_n \cdot \mathbf{A}$

The pattern in this formula can be made a bit more explicit. Let us use the symbol $\circ$ to denote component-wise matrix multiplication, and think about this in terms of block matrix multiplication where $\mathbf{A}_{\text{even}}$ is a column matrix of the coefficients of A_{even}, and

$$\mathbf{F}_{2k} \cdot \mathbf{A} = \begin{pmatrix} \mathbf{F}_k \cdot \mathbf{A}_{\text{even}} \\ \mathbf{F}_k \cdot \mathbf{A}_{\text{even}} \end{pmatrix} + \begin{pmatrix} 1 \\ \omega \\ \omega^2 \\ \cdots \\ w^{2k-1} \end{pmatrix} \circ \begin{pmatrix} \mathbf{F}_k \cdot \mathbf{A}_{\text{odd}} \\ \mathbf{F}_k \cdot \mathbf{A}_{\text{odd}} \end{pmatrix}$$

We have accomplished a 2k Fourier Transform with 2 size k Fourier Transforms plus some O(n) operations and powerings of ω (which can be precomputed).

How fast is this FFT?

A principal item of interest is the running time for the FFT. If the time to perform an FFT of size 2k is T(2k), then from our expression above,

$T(2k)=2T(k)+O(k)$.
Iterating this formula gives (where $r=\log_2 k$):

$T(2k)=2^r T(1) + O(rk)= O(n \log n)$.

Thus this trick reduces the $O(n^2)$ algorithm (of n applications of Horner's rule), to $O(n \log n)$ operations. This is a very good speedup, and was allegedly known by Gauss and others, until it was popularized by Cooley and Tukey.

We have to be able to invert the FFT = Interpolation!

That is, we are given a set of evaluation points (powers of a root of unity) and asked to provide a polynomial which assumes given values at those points.
The inverse of F_n is a matrix G_n
where the $(i,j)^{th}$ element is ω_n^{-ij} /n,
where we have, somewhat unconventionally, numbered the rows and columns so that the 0,0 element is at the top left.

The Inverse of F_n

$$\mathbf{G}_n = \frac{1}{n}\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \dots & \omega^{-(n-1)} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \dots & \omega^{-(n-1)^2} \end{pmatrix}$$

It looks rather like F_n with ω^{-1} in place of ω, and with a 1/n out front. So whatever computation structure we have needs to allow us to compute 1/n, as well as have roots of unity we can use.

Proof that $I = F_n G_n$

in the i, jth position of $\mathbf{F} \cdot \mathbf{G}$ we have

$$\frac{1}{n} \sum_{k=0}^{n-1} \omega^{ik} \omega^{-jk}.$$

If $i = j$ this is just 1, and if $h = i - j \neq 0$, we have the geometric series

$$\frac{1}{n} \sum_{k=0}^{n-1} \omega^{hk} = \frac{1}{n} \frac{(\omega^h)^n - 1}{(\omega^h) - 1} = 0.$$

It's zero because $(\omega^h)^n - 1 = (\omega^n)^h - 1 = 1^h - 1 = 0$.

How hard is it to find primitive roots of unity? Not hard.

There is a 2^r th root of unity in a field $\mathbf{Z}_m$ (m prime) if $m = c2^r + 1$. E.g. 3 is a primitive 8th root of 1 in $\mathbf{Z}_{41}$.

How many of these can we find?

There are about 50 primes $m < 2^{31}$ of the form $c2^r + 1$ where $r \leq 20$. That is, even among the "single-computer-word" primes, there are plenty of primes of that form.

Thus if we wish to use an FFT, we can have a pre-computed list of 2^{20} th roots of unity (allowing about a 10^6 point transform). Moreover, $\omega_{2^{19}}, \omega_{2^{18}}$, etc. are trivially computed by squaring $\omega_{2^{20}}$, etc. so that we have roots of all smaller power-of-two sizes.

The algorithm, written simply, recursively

Input: k, A power of 2.

$\mathbf{A}$, a sequence of length k of elements in $\mathbf{S}$, a finite computation structure.

ω_k, a kth root of unity in $\mathbf{S}$.

Output: $\hat{\mathbf{A}}$

The insides:

0. If $k = 1$ then return $\hat{\mathbf{A}} = \mathbf{A}$.

1. [split] $\mathbf{B} := even(\mathbf{A})$; $\mathbf{C} := odd(\mathbf{A})$; {Separating the original sequence into two sub-sequences}.

2. [recursive steps] $\hat{\mathbf{B}} :=$ FFT$(k/2, \mathbf{B}, \omega_k^2)$;

   $\hat{\mathbf{C}} :=$ FFT$(k/2, \mathbf{C}, \omega_k^2)$

3. For i:=0 to k/2-1 do

   begin

   $\hat{\mathbf{A}}_i := \hat{\mathbf{B}}_i + \omega_k^i \hat{\mathbf{C}}_i$;

   $\hat{\mathbf{A}}_{i+k/2} := \hat{\mathbf{B}}_i + {\omega_k}^{i+k/2} \hat{\mathbf{C}}_i$

   end

Can the FFT be improved?

Basically, it continues to be a major industry to come up with improved designs, variations and implementations of the FFT.

http://www.fftw.org/

is the home page for the fastest FFT "in the West".

Limitations that can be modified: power of 2, (vs power of 3 or product-of primes) in-place transforms, cache performance.

Since I usually badmouth asymptotic results, is this practical (e.g. for polynomials)?

Yes: Enough of the time that it has been implemented in a few systems.

No: It is not the default for any computer algebra system. Probably not fast enough for routinely-small routinely-sparse routinely many-variables. Coefficients must be bounded in some effective way.

Other techniques (e.g. Karatsuba) sometimes prevail.

Benchmark

One benchmark set up to favor FFTs suggest that computing $f(x)^2$ where f is a dense polynomial over a single-precision-size finite field, is best done by FFT only if deg(f)>96.

How favored?

1. $(x+1)^{96}$ has many "bignum" coefficients, so a real comparison might need 3 or 4 FFTs + CRA + bound computation.
2. For f(x) sparse, FFTs would do almost as much work, but the competition would not.