

## Lecture 4: 09/17/01

Lecturer: Micah Adler

Scribe: Rob Platt and Bryan Thibodeau

## 4.1 Fast Fourier Transforms (F.F.T.)

### 4.1.1 Applications of the F.F.T.

The FFT has a broader usefulness than just polynomial multiplication. Here are some examples:

1. signal processing.
2. coding theory.
3. Schönhage-Strassen Algorithm. This algorithm is used for the multiplication of 2  $n$ -bit integers. This algorithm requires  $\Theta(n \log n \log \log n)$  bit operations and is the fastest known algorithm to solve the problem. For more details about the algorithm see [AHU74, pp 270–274].

### 4.1.2 Recap of the game plan from last time

We will use the Fast Fourier Transform (FFT) to convert back and forth between the point-value representation and the coefficient representation (called evaluation and interpolation respectively). Using the naive approach the evaluation of a polynomial at  $n$  points will require  $\Theta(n^2)$  time. But if we choose the evaluation points as the  $n$  complex  $n$ th roots of unity we can achieve a better performance.

The steps to multiply two polynomials using F.F.T are diagramed below:

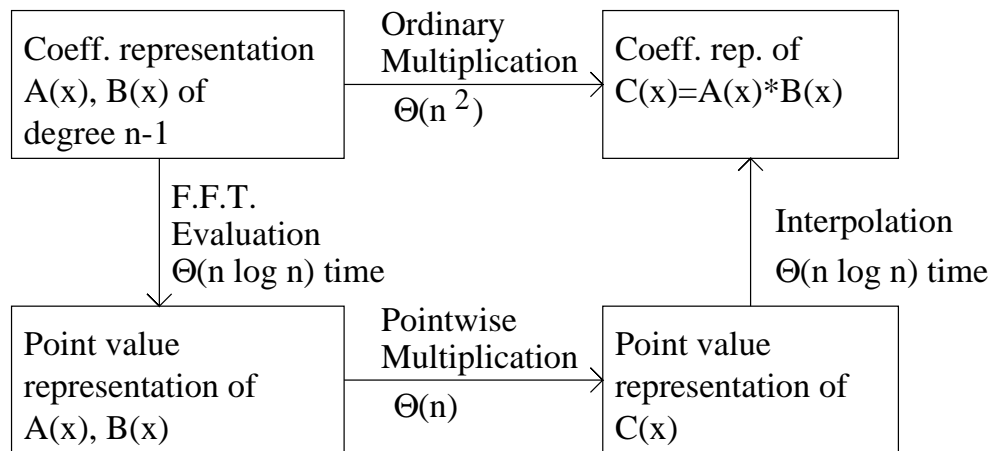


Figure 4.1: A Graphical Outline of Polynomial Multiplication using F.F.T. [VD00]

### 4.1.3 Interpolation at the complex roots of unity

So far we have seen the evaluation of the polynomial at the complex roots of unity using the Fast Fourier Transform (F.F.T.). But for the polynomial multiplication to be complete we must have an algorithm that can be used to convert the point-value representation of the polynomial to the coefficient form. This conversion is called the interpolation [VD00].

Let this be the polynomial we are working with:

$$A(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$$

$y_i = A(\omega_n^i)$  defines the point-value representation of the polynomial as:

$$\{(\omega_n^0, y_0), (\omega_n^1, y_1), \dots, (\omega_n^{n-1}, y_{n-1})\}$$

In the evaluation step, we are given:  $a_0, a_1, \dots, a_{n-1}$ . We must compute:  $y_0, y_1, \dots, y_{n-1}$ .

In the interpolation step, we are given  $y_0, y_1, \dots, y_{n-1}$ , and we are asked to compute:  $a_0, a_1, \dots, a_{n-1}$ . Let us examine how to do this:

We can write the evaluation of the polynomial at the  $n$ th roots of unity as the matrix product  $\mathbf{y} = \mathbf{V}_n \cdot \mathbf{a}$  where  $\mathbf{V}_n$  is a Vandermonde matrix containing the appropriate powers of  $\omega_n$  [VD00]:

$$\mathbf{V}_n = \begin{pmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \dots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \dots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \dots & \omega_n^{(n-1)(n-1)} \end{pmatrix}$$

The  $(k, j)$  entry of  $\mathbf{V}_n$  is  $\omega_n^{kj}$  for  $j, k = 0, 1, 2, \dots, n-1$ .

$$\mathbf{y} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

$$\mathbf{a} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

The inverse operation can be written as  $\mathbf{a} = \mathbf{V}_n^{-1} \cdot \mathbf{y}$  where:

$$\mathbf{V}_n^{-1} = \frac{1}{n} * \begin{pmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \omega_n^{-3} & \dots & \omega_n^{-(n-1)} \\ 1 & \omega_n^{-2} & \omega_n^{-4} & \omega_n^{-6} & \dots & \omega_n^{-2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \omega_n^{-3(n-1)} & \dots & \omega_n^{-(n-1)(n-1)} \end{pmatrix}$$

Thus the  $(j, k)$  entry of  $\mathbf{V}_n^{-1}$  is  $\omega_n^{-kj}/n$  for  $j, k = 0, 1, 2, \dots, n-1$ .

As an exercise, verify that  $\mathbf{V}_n^{-1}$  is indeed the inverse by showing that  $\mathbf{V}_n^{-1} \cdot \mathbf{V}_n = \mathbf{I}$ .

Thus the algorithm for the interpolation can be informally written as:

```

if  $n = 1$ , return  $y_0$ 

else //divide and conquer

     $a^{even} = F.F.T.(y_0, y_2, y_4, \dots, y_{n-2})$ 
     $a^{odd} = F.F.T.(y_1, y_3, y_5, \dots, y_{n-1})$ 

    for  $k = 0$  to  $\frac{n}{2} - 1$  //combine
         $a_k = a_k^{even} + \omega_n^{-k} a_k^{odd}$ 
         $a_{k+\frac{n}{2}} = a_k^{even} - \omega_n^{-k} a_k^{odd}$ 
    return  $\frac{a}{n}$ 

```

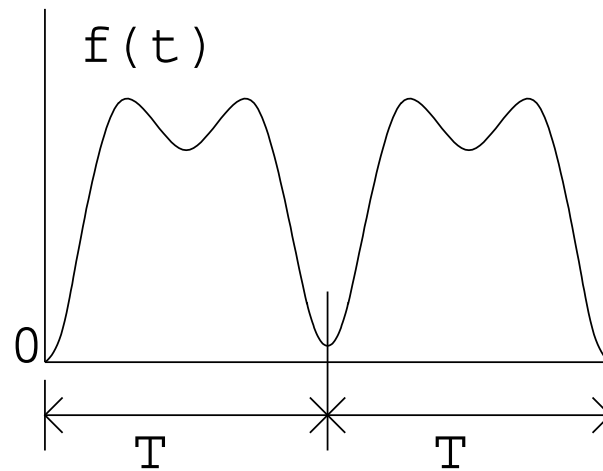
Which is just the F.F.T. algorithm with three modifications [VD00]:

1. switch roles of  $y$  and  $a$
2. replace  $\omega_n$  by  $\omega_n^{-1}$
3. divide result by  $n$

Thus the interpolation can be done in  $\Theta(n \log n)$  time as well. This interpolation is also called the inverse F.F.T.. Thus by using the F.F.T. and the Inverse F.F.T. we can transform a polynomial of degree bound  $n$  back and forth between its coefficient and point-value representation in time  $\Theta(n \log n)$ . Thus we have shown that we can multiply two polynomials in  $\Theta(n \log n)$  time. This is better than the  $\Theta(n^2)$  time of the naive approach [VD00].

#### 4.1.4 What is a Fourier Transform?

Let  $f(t)$  be any periodic function as shown in the figure below.

Figure 4.2: A periodic function  $f(t)$  [VD00]

$f(t)$  can also be expressed as a linear sum of elementary periodic functions with periods  $\inf, T, T/2, T/3, \dots$ . The elementary periodic functions are shown below.

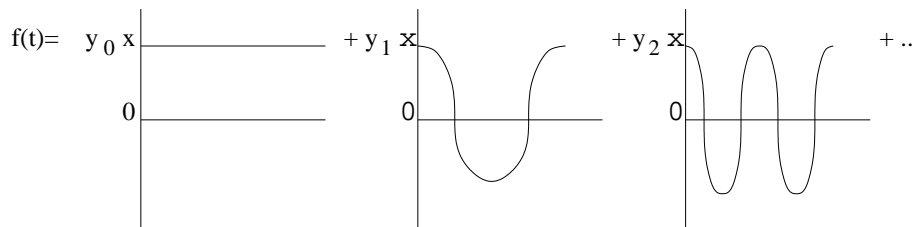


Figure 4.3: F.F.T. as a sum of elementary periodic functions [VD00]

$y_0, y_1, y_2 \dots$  are known as the Fourier coefficients.

In contrast to the Fourier Transform, the Discrete Fourier Transform evaluates the periodic function  $f(t)$  at  $n$  evenly spaced discrete points every period. Thus if the coefficient vector of  $f(t)$  is  $a = (a_0, a_1, \dots, a_{n-1})$  and if  $y = (y_0, y_1, \dots, y_{n-1})$  are the values of  $f(t)$  at the  $n$  evenly spaced values of  $t$ , then we write  $y = DFT(a)$ . An interesting characteristic of DFTs is that if the function  $f(t)$  is evaluated at  $n$  discrete points per period, then only the first  $n$  coefficients are required to completely characterize the function at those points.

## 4.2 Kruskal's Algorithm for computing a minimum spanning tree

Kruskal's Algorithm is an example of a greedy algorithm, i.e. an algorithm that will always take the best current choice without regard to future consequences.

**Input:** Undirected graph  $G = (V, E)$  with edge weights.

**Output:** A spanning tree (a connected subset of edges with no cycle) of  $G$  with minimum total weight.

### 4.2.1 The Algorithm

Maintain a forest  $F$  (an acyclic subgraph of  $G$ )

Sort edges by increasing weight

$F = \emptyset$

Do until  $F$  is a spanning tree of  $G$ :

    get the next edge  $e$

    if  $F + e$  is acyclic then  $F = F + e$

Return  $F$

### 4.2.2 Run Time

1. **Sorting:**  $\Theta(|E| \log |E|)$
2. **Determining if  $F + e$  is cyclic:** Assume that we are implementing the algorithm by using an array for each vertex, and each entry in the array is a label that denotes which connected component the vertex belongs to. Now to check if  $F + e$  is cyclic we just need to check the entries in the array for the two vertices that  $e$  connects. If they are the same then  $F + e$  is cyclic, if they are different the  $F + e$  is acyclic. The runtime to perform this check is  $\Theta(1)$ .
3. **Adding  $e$  to  $F$ :** To do this we first need to switch the labels for the smaller connected component. This takes  $O(|V|)$  time. This step can be performed at most  $|V|$  times, so it takes at most  $\Theta(|V|^2)$  to add the edges to the forest.

Thus the total run time of the algorithm is  $O(|V|^2 + |E| \log |E|)$ . Algorithms exist that can compute a minimum spanning tree in  $O(|E| \log |E|)$ .

### 4.2.3 Proof of Correctness

**Claim 4.1** *Given any unconnected forest  $F$ , let  $S(F)$  be the set of spanning trees that include  $F$ .*

*There is a minimum cost spanning tree  $T \in S(F)$  that includes edge  $e$ , the minimum weight edge that is: (a) not in  $F$ , and (b) does not cause a cycle with  $F$ .*

**Proof:** Assume that the claim is not true and there exists some  $T' \in S(F)$  such that  $e \notin T'$ , and the cost of  $T' <$  the cost of any tree containing  $e$ .

Add  $e$  to  $T'$  and a cycle results. (See fig 4.4)

There must be some edge  $e'$  that is in the cycle that was not in the original forest  $F$ .

There is some edge  $e' \neq e$  on the cycle such that  $e' \notin F$

Weight of  $e' >$  weight of  $e$ .

Weight of  $T' + e - e' \leq$  weight of  $T'$ . This contradicts our assumption, therefore the claim is true. ■

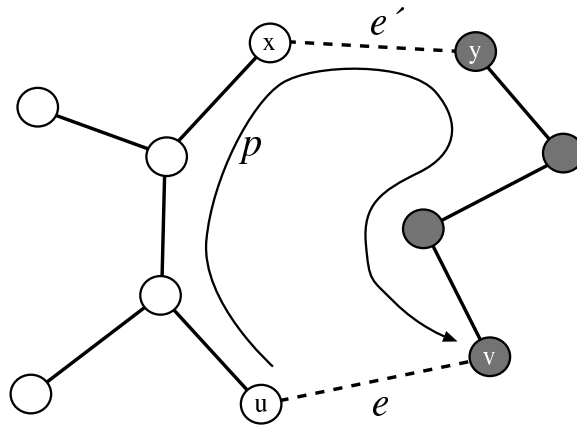


Figure 4.4: Depiction of forest  $F$  in  $S[\text{MH00}]$

**Proof:** We can show by induction that Kruskal's Algorithm is correct.

**Base Case:**  $F$  is empty

**Inductive Step:** At every step, Kruskal's Algorithm will add the minimum weight edge that is not in  $F$  and that does not cause a cycle in  $F$ . Claim 4.1 tells us then that at every step,  $F$  will only contain minimum spanning trees. Thus when the algorithm finishes  $F$  will be a minimum spanning tree of  $G$ . ■

## References

- AHU74 A. AHO, J. HOPCROFT, and J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
- MU00 MORRISON, and HUANG, *Scribe Notes for Lecture 4*, UMASS course 611: Advanced Algorithms, Fall 2000.
- VD00 VASUDEVAN, and DICKSON, *Scribe Notes for Lecture 3*, UMASS course 611: Advanced Algorithms, Fall 2000.