UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Emprego de FFT para convoluções em funções de domínio discreto

Guilherme Hideo Tubone

São Carlos – SP

Emprego de FFT para convoluções em funções de domínio discreto

Guilherme Hideo Tubone

Orientador: **Prof. Dr. João do E.S. Batista Neto**

Monografia final de conclusão de curso apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como requisito parcial para obtenção do título de Bacharel em Computação.
Área de Concentração: Algoritmos

USP – São Carlos
Junho de 2019

Tubone, Guilherme Hideo
Emprego de FFT para convoluções em funções de domínio discreto / Guilherme Hideo Tubone. - São Carlos - SP, 2019.
35 p.; 29,7 cm.

Orientador: João do E.S. Batista Neto.
Monografia (Graduação) - Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos - SP, 2019.

1. Transformada Rápida de Fourier. 2. FFT. 3. DFT. 4. Convolução. 5. Polinômios. I. Neto, João do E.S. Batista. II. Instituto de Ciências Matemáticas e de Computação (ICMC/USP). III. Título.

AGRADECIMENTOS

Agradeço à minha família, por todo apoio e suporte dado ao longo de toda a minha vida.

Agradeço também ao meu orientador, pela oportunidade de desenvolver este trabalho e pela confiança depositada em mim.

E por fim, agradeço ao GEMA(Grupo de Estudos para Maratona de Programação) e à própria Maratona, que me fizeram evoluir muito em relação aos estudos de computação, me trouxeram grandes amigos e me apresentaram inúmeras oportunidades ao longo da graduação.

RESUMO

TUBONE, G. H.. **Emprego de FFT para convoluções em funções de domínio discreto**. 2019. 35 f. Monografia (Graduação) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

A Transformada Rápida de Fourier (FFT) é um algoritmo que eficientemente calcula a Transformada discreta de Fourier (DFT) e a sua inversa. Esta transformada converte sinais de seu domínio original (muitas vezes espaço ou tempo) para o domínio de frequências e vice-versa. Algumas aplicações conhecidas para o FFT são: processamento de sinais digitais, resolução de equações diferenciais, multiplicações de polinômios e números grandes, entre outras. Este artigo visa mostrar algumas aplicações para o algoritmo de FFT em situações não-convencionais, utilizando como exemplos problemas de programação competitiva, na qual é muito comum utilizar este algoritmo para realizar convoluções em funções de domínio discreto.

Palavras-chave: Transformada Rápida de Fourier, FFT, DFT, Convolução, Polinômios.

LISTA DE ILUSTRAÇÕES

LISTA DE ALGORITMOS

SUMÁRIO

Capítulo 1

INTRODUÇÃO

O primeiro registro da utilização da Transformada Rápida de Fourier (FFT) é de meados de 1805, quando Gauss a usou para interpolar as órbitas dos asteróides Pallas e Juno. Entretanto, o FFT foi popularizado por Cooley e Tukey apenas em 1965, quando estes publicaram uma versão mais moderna e geral do algoritmo (HEIDEMAN; JOHNSON; BURRUS, 1985). Algumas aplicações importantes deste algoritmo são: multiplicacão rápida de números grandes e polinômios, resolução de equações diferencias ordinárias, algoritmos de aplicação de filtros, entre outras (ROCKMORE, 2000).

A inspiração para a escrita deste artigo foi a Maratona de Programação, um evento anual de programação competitiva organizado pela Sociedade Brasileira de Computação. Nela, é comum aparecerem problemas relacionados com convoluções em funções de domínio discreto, e, portanto, se faz necessária a utilização da Transformada Rápida de Fourier (FFT) para resolvê-los.

O objetivo deste trabalho é ter como resultado um tutorial para alunos de computação interessados em aprender novos algoritmos, em especial para aqueles que participam de competições de programação e desejam melhorar o seu desempenho nestes tipos de competição.

Iniciaremos o artigo falando sobre representações de polinômios (coeficientes e ponto-valor) e discutindo as complexidades de realizar algumas operações com eles em cada uma das representações. Perceberemos nesta seção, que algumas operações são realizadas com muito mais eficiência dependendo de como o polinômio está representado. Logo após, discutiremos algumas formas conhecidas de converter polinômios de uma representação para outra, com o objetivo aproveitar o que há de melhor em cada uma das representações.

Então, apresentaremos a Transformada Rápida de Fourier (FFT), que é um algoritmo capaz de realizar a transição de um polinômio entre as formas de coeficientes e ponto-valor. Deste modo, conseguiremos multiplicar polinômis em complexidade $O(n \cdot \log_2 n)$, ao invés da solução trivial, que leva $O(n^2)$.

Por fim, deduziremos que realizar uma convolução entre duas funções de domínio discreto é equivalente a multiplicar dois polinômios. Assim, utilizar FFT para realizar multiplicação entre polinômios se mostrará ser uma técnica poderosa para realizar convoluções entre funções de domínio discreto, como mostraremos com exemplos de problemas no último capítulo do artigo.

Capítulo 2

POLINÔMIOS

2.1 Representação de polinômios

Um polinômio $A(x)$ de grau $n-1$ pode ser representado na sua forma de **coeficientes** da seguinte maneira, onde $a_0, a_1, a_2, ..., a_{n-1}$ são os chamados coeficientes do polinômio:

$$\begin{aligned} A(x) &= a_0 + a_1 x^1 + a_2 x^2 + ... + a_{n-1} x^{n-1} \\ &= \sum_{i=0}^{n-1} a_i x^i \end{aligned}$$

Uma outra maneira de representar um polinômio $A(x)$ é na forma de **ponto-valor**. Para um polinômio de grau $n-1$, escolhemos n pontos $x_0, x_1, x_2, ..., x_{n-1}$ distintos e suas respectivas valorações:

$$\begin{aligned} &(x_0, A(x_0)) \\ &(x_1, A(x_1)) \\ &(x_2, A(x_2)) \\ &... \\ &(x_{n-1}, A(x_{n-1})) \end{aligned}$$

De acordo com a álgebra, isto determina unicamente um polinômio interpolador de grau $n-1$ (FRANCO, 2006).

2.2 Complexidade de operações

Nesta seção, discutiremos como realizar computacionalmente algumas operações com polinômios e as respectivas complexidades assintóticas de seus algoritmos nas duas representações anteriormente apresentadas (coeficientes e ponto-valor). Isto servirá principalmente para entendermos os prós e contras de termos um polinômio representado em cada uma destas duas formas.

2.2.1 Soma

Dados dois polinômios $A(x)$ e $B(x)$, ambos de grau $n-1$, queremos obter $C(x) = A(x) + B(x)$.

Coeficientes

Realizamos a soma dos coeficientes termo-a-termo:

$$c_i = a_i + b_i, \forall 0 \leq i \leq n-1$$

A complexidade para esta operação é $O(n)$, pois existem n coeficientes em cada polinômio.

Ponto-valor

Realizamos a soma das valorações $A(x_0), A(x_1), A(x_2), ..., A(x_{n-1})$ com as valorações $B(x_0), B(x_1), B(x_2), ..., B(x_{n-1})$ nos seus pontos correspondentes, portanto:

$$C(x_i) = A(x_i) + B(x_i), \forall 0 \leq i \leq n-1$$

A complexidade para esta operação é $O(n)$, pois existem n pares (x_i, y_i).

2.2.2 Multiplicação

Dados dois polinômios $A(x)$ e $B(x)$, ambos de grau $n-1$, queremos obter $C(x) = A(x) \cdot B(x)$.

É importante observar que o produto dos polinômios possuirá grau $2n-2$, já que estamos multiplicando dois polinômios de grau $n-1$.

Coeficientes

Temos que aplicar a propriedade distributiva para todo termo de $A(x)$. Ou seja, é necessário fazer o produto de todo termo de $A(x)$ com todo termo de $B(x)$, portanto, podemos fazer:

$$c_i = \sum_{j=0}^{i} a_j \cdot b_{i-j}, \forall 0 \leq i \leq 2n-2$$

A complexidade para esta operação é $O(n^2)$, pois cada um dos n termos de A(x) será multiplicado por todos os n termos de B(x).

Ponto-valor

Primeiramente, é necessário que façamos uma extensão dos pares de ponto-valor dos polinômios $A(x)$ e $B(x)$, de tal forma que tenhamos $2n-1$ pares distintos para cada um, pois o polinômio resultante terá grau $2n-2$. Agora, de forma similar à soma, fazemos:

$$C(x_i) = A(x_i) \cdot B(x_i), \forall 0 \leq i \leq 2n-2$$

A complexidade para esta operação é $O(n)$, pois existem $2n-1$ pares (x_i, y_i).

2.2.3 Valoração

Dado um polinômio $A(x)$, de grau $n-1$, queremos obter seu valor no ponto x_0.

Coeficientes

Basta realizarmos as operações aritméticas necessárias para o cálculo, levando em consideração que não precisamos calcular x^i em $O(i)$ para cada termo, visto que podemos reaproveitar o valor de x^{i-1} fazendo: $x^i = x^{i-1} \cdot x$.

A complexidade para esta operação é $O(n)$, pois o polinômio possui n termos.

Ponto-valor

Para avaliar um polinômio em x_o na forma de ponto-valor é necessário que previamente façamos uma interpolação do mesmo, por exemplo, utilizando a Fórmula de Lagrange.

A complexidade para esta operação é $O(n^2)$.

2.2.4 Conversão entre forma de coeficientes e ponto-valor

Como podemos observar, para somar dois polinômios temos uma complexidade linear que se mantém entre as duas formas. Entretanto, multiplicar dois polinômios tem complexidade quadrática na forma de coeficientes e complexidade linear na forma de ponto-valor. Enquanto que na valoração o que acontece é o oposto, temos complexidade linear na forma de coeficientes, porém quadrática na forma de ponto-valor.

Portanto, para aproveitarmos o que há de melhor em cada uma das representações, precisamos de um meio eficiente para converter um polinômio da sua forma de coeficientes para a forma de ponto-valor e vice-versa.

Analisemos a seguinte relação:

$$V \cdot A = Y$$

$$\begin{bmatrix} x_0^0 & x_0^1 & x_0^2 & \cdots & x_0^{n-1} \\ x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^{n-1} \\ x_2^0 & x_2^1 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{n-1}^0 & x_{n-1}^1 & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

Aqui, V é a chamada Matriz de Vandermonde, em que as linhas são referentes a pontos x_j distintos e cada coluna representa as potências de 0 a $n-1$ de cada ponto, A é o vetor dos coeficientes a_i de um polinômio $A(x)$ e Y é o vetor das valorações y_j em $A(x)$ para cada um dos pontos x_j.

Desta forma, para convertemos da representação de coeficientes para ponto-valor, basta fazermos $Y = V \cdot A$, que tem complexidade $O(n^2)$.

Já para convertermos da representação de ponto-valor para a de coeficientes, é necessário resolver o seguinte sistema linear:

$$\begin{bmatrix} x_0^0 \cdot a_0 & x_0^1 \cdot a_1 & x_0^2 \cdot a_2 & \cdots & x_0^{n-1} \cdot a_{n-1} \\ x_1^0 \cdot a_0 & x_1^1 \cdot a_1 & x_1^2 \cdot a_2 & \cdots & x_1^{n-1} \cdot a_{n-1} \\ x_2^0 \cdot a_0 & x_2^1 \cdot a_1 & x_2^2 \cdot a_2 & \cdots & x_2^{n-1} \cdot a_{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{n-1}^0 \cdot a_0 & x_{n-1}^1 \cdot a_1 & x_{n-1}^2 \cdot a_2 & \cdots & x_{n-1}^{n-1} \cdot a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

Algumas das formas possíveis para resolvê-lo em complexidade $O(n^3)$ são através da Eliminação Gaussiana ou da Decomposição LU (FRANCO, 2006).

Nos próximos capítulo mostraremos como estas duas conversões podem ser feitas de maneira mais eficiente utilizando a **Transformada Rápida de Fourier (FFT)** e, consequentemente, como multiplicar dois polinômios na forma de coeficientes pode ser feito de maneira mais eficiente.

Capítulo 3

TRANSFORMADA RÁPIDA DE FOURIER (FFT)

3.1 Divisão e Conquista

Nesta seção, mostraremos um algoritmo de Divisão e Conquista que tem como finalidade converter um polinômio da forma de coeficientes para a forma de ponto-valor, ou seja, avaliar um polinômio $A(x)$ de grau $n-1$ em n pontos distintos $x \in X$. Veremos também que esta solução por si só ainda não será a mais eficiente para resolver este problema, mas que ela é essencial para a compreensão dos próximos passos.

O algoritmo consiste em uma recursão que, a cada chamada, cria dois novos polinômios a partir do polinômio em questão e, recursivamente, os avalia em um novo conjunto de pontos $X^2 = \{x^2 | x \in X\}$, a fim de obter as valorações do primeiro polinômio a partir das valorações dos dois novos polinômios. A seguir descreveremos com mais detalhes as três etapas deste algoritmo: Divisão, Conquista e Combinação.

3.1.1 Divisão

Nesta etapa, dividiremos os coeficientes do polinômio $A(x)$ para criar dois novos polinômios: o dos coeficientes pares e o dos ímpares. Desta forma, o grau dos dois novos polinômios são diminuídos e os seus coeficientes rearranjados para outros termos:

$$A_{par}(x) = a_0 + a_2 x + a_4 x^2 + ... + a_{2\lceil n/2-1 \rceil} x^{\lceil n/2-1 \rceil} = \sum_{i=0}^{\lceil n/2-1 \rceil} a_{2i} x^i$$

$$A_{impar}(x) = a_1 + a_3 x + a_5 x^2 + ... + a_{2\lfloor n/2-1 \rfloor} x^{\lfloor n/2-1 \rfloor} = \sum_{i=0}^{\lfloor n/2-1 \rfloor} a_{2i+1} x^i$$

Note que para rearranjar os coeficientes para o seus termos originais, basta fazermos:

$$A_{par}(x^2) = a_0 + a_2 x^2 + a_4 x^4 + ... + a_{2\lceil n/2-1 \rceil} x^{\lceil n/2-1 \rceil^2}$$

$$A_{impar}(x^2) \cdot x = a_1 x + a_3 x^3 + a_5 x^5 + ... + a_{2\lfloor n/2-1 \rfloor} x^{\lfloor n/2-1 \rfloor^2}$$

3.1.2 Conquista

Agora que dividimos o polinômio, temos dois novos sub-problemas, que serão resolvidos recursivamente, com a diferença que desta vez queremos avaliar os novos polinômios para todo $y \in X^2$, onde $X^2 = \{x^2 | x \in X\}$.

3.1.3 Combinação

Já vimos na parte de Divisão como rearranjar os termos dos dois novos polinômios para a configuração do polinômio original. Portanto, para combinar as respostas destes dois polinômios e formar a resposta do polinômio original basta fazermos:

$$A(x) = A_{par}(x^2) + x \cdot A_{impar}(x^2)$$

$$A(x) = (a_0 + a_2x^2 + ... + a_{2\lceil n/2-1 \rceil}x^{\lceil n/2-1 \rceil^2}) + (a_1x + a_3x^3 + ... + a_{2\lfloor n/2-1 \rfloor}x^{\lfloor n/2-1 \rfloor^2})$$

$$A(x) = a_0 + a_1x^1 + a_2x^2 + a_3x^3 + ... + a_{n-1}x^{n-1}$$

3.1.4 Algoritmo e análise de complexidade

Em seguida, descreveremos, em pseudocódigo, a função que implementa o algoritmo de Divisão e Conquista. Ela recebe como parâmetro dois vetores: A, dos coeficientes do polinômio e X, do conjunto de pontos a serem avaliados no polinômio.

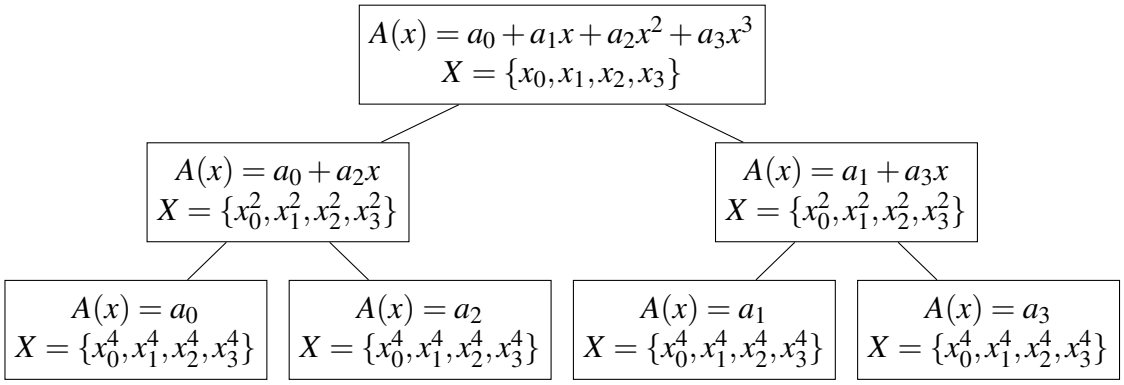
Para facilitar a implementação do algoritmo, previamente, iremos alterar o tamanho do vetor de coeficientes para a menor potência de 2 que é maior ou igual ao tamanho atual do vetor. Desta forma, neste código, podemos assumir que o vetor A sempre terá tamanho 2^i.

Algoritmo 1: Algoritmo de Divisão e Conquista para avaliar um polinômio em n pontos

```
1 Divisao-E-Conquista(A, X)
2 n = A.tamanho                                  // Sempre é potência de 2
3 if n == 1 then
4     return (A_0, A_0, ..., A_0)                // Vetor de tamanho n
5 A_par = (A_0, A_2, ..., A_{n-2})
  A_impar = (A_1, A_3, ..., A_{n-1})
6 X^2 = X * X
7 Y_par = Divisao - E - Conquista(A_par, X^2)
  Y_impar = Divisao - E - Conquista(A_impar, X^2)
8 return Y_par + X * Y_impar
```

Segue uma representação da árvore de recursão do algoritmo para um exemplo em que temos um polinômio de grau 3 e queremos avaliá-lo em 4 pontos x_0, x_1, x_2, x_3:

Figura 1 – Árvore de recursão do algoritmo de divisão e conquista



Observemos que, a cada chamada recursiva da função, o tamanho do vetor de coeficientes diminui pela metade, entretanto, o tamanho do conjunto X se mantém o mesmo. Ou seja, para cada nó da árvore de recursão do algoritmo (de um total de $2n-1$), o polinômio referente a este nó terá que ser avaliado nos n pontos do conjunto X. Portanto, este algoritmo segue a seguinte recorrência:

$$T(n,|X|) = 2T(\frac{n}{2},|X^2|) + O(n+|X|)$$
$$T(n,|X|) = 2T(\frac{n}{2},|X|) + O(n+|X|)$$
$$T(n,|X|) = O(n^2)$$

3.2 Transformada Rápida de Fourier

No algoritmo descrito anteriormente, percebemos que o gargalo da sua execução está no fato de que o tamanho do conjunto de pontos X se mantém o mesmo para toda chamada recursiva. Este problema seria resolvido caso o tamanho do conjunto X diminuísse pela metade, assim como acontece com o tamanho do vetor de coeficientes. Pois, desta forma, para um nó de profundidade p da árvore de recursão, o polinômio deste nó teria que ser avaliado apenas em $\frac{n}{2^p}$ pontos.

Nesta seção, mostraremos o que são as raízes complexas da unidade, algumas de suas propriedades e como estas podem fazer com que o nosso atual algoritmo se torne muito mais eficiente.

3.2.1 Raízes da unidade

A partir deste momento, sempre que utilizarmos a letra i, será exclusivamente para representar o número imaginário $i = \sqrt{-1}$.

Uma n-ésima raiz da unidade é um número complexo z que satifaz a relação $z^n = 1$ e pode ser representada da seguinte maneira, para todo $k \in \mathbb{Z}$, com $0 \leq k < n$:

$$\begin{aligned} z &= e^{\frac{i2\pi k}{n}} \\ &= \cos(\frac{k}{n}2\pi) + i \cdot \sin(\frac{k}{n}2\pi) \text{ (Fórmula de Euler)} \end{aligned}$$

Demonstração.

$$\begin{aligned} z &= e^{\frac{i2\pi k}{n}} \\ z^n &= e^{(\frac{i2\pi k}{n})^n} \\ &= e^{\frac{i2\pi kn}{n}} = e^{i2\pi k} \\ &= \cos(2\pi k) + i \cdot \sin(2\pi k) \\ &= 1 + i \cdot 0 = 1 \end{aligned}$$

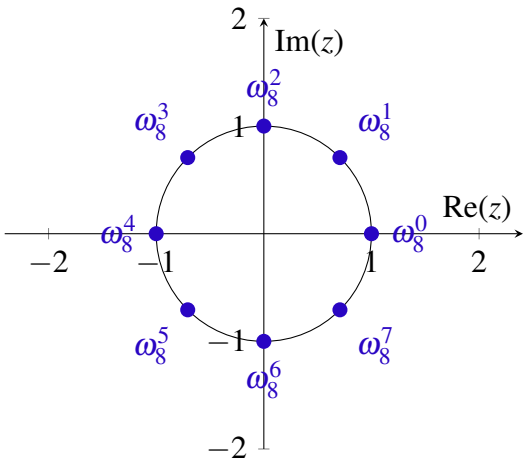
□

Denotaremos como a n-ésima raiz principal $\omega_n = e^{\frac{i2\pi}{n}}$. As outras n-ésimas raízes serão sempre potências da raiz principal: $\omega_n^1, \omega_n^2, ..., \omega_n^{n-1}$.

Uma observação importante é que, geometricamente, estes números estão contidos no círculo de raio unitário, centrado na origem do plano complexo. Pois, pela relação fundamental da trigonometria, o argumento |z| das raízes é igual a 1:

$$\begin{aligned} |z| &= \sqrt{\cos^2\theta + \sin^2\theta} \\ |z| &= \sqrt{1} = 1 \end{aligned}$$

Figura 2 – 8-ésimas raízes da unidade representadas no círculo unitário



Notemos também que as raízes estão igualmente espaçadas no círculo. Isto porque, de forma intuitiva, estamos dividindo o círculo em n partes e cada um dos valores está exatamente na divisão entre estas partes (i.e., o ângulo da k-ésima parte é $\frac{2\pi k}{n}$).

Agora, mostraremos algumas propriedades das raízes da unidade (CORMEN *et al.*, 2009) que serão importantes para tornar o algoritmo anterior mais eficiente:

Lema 1. (do Cancelamento). Para quaisquer inteiros $n \geq 0$, $k \geq 0$ e $d > 0$, $\omega_{dn}^{dk} = \omega_n^k$.

Demonstração.

$$\begin{aligned} \omega_{dn}^{dk} &= \omega_n^k \\ e^{\frac{i2\pi dk}{dn}} &= e^{\frac{i2\pi k}{n}} \\ e^{\frac{i2\pi k}{n}} &= e^{\frac{i2\pi k}{n}} \\ 1 &= 1 \end{aligned}$$

□

Lema 2. Para qualquer inteiro $n > 0$, $\omega_n^{\frac{n}{2}} = \omega_2 = -1$.

Demonstração.

$$\begin{aligned} \omega_n^{\frac{n}{2}} &= e^{\frac{i2\pi(\frac{n}{2})}{n}} \\ &= e^{\frac{in\pi}{n}} = e^{i\pi} \\ &= \cos(\pi) + i \cdot \sin(\pi) = -1 + i \cdot 0 \\ &= -1 \end{aligned}$$

□

Lema 3. (da Metade). Se $n > 0$ é par, então os quadrados das n n-ésimas raízes da unidade são as $\frac{n}{2}$ ($\frac{n}{2}$)-ésimas raízes da unidade. Ou seja, $(\omega_n^k)^2 = (\omega_n^{k+\frac{n}{2}})^2 = \omega_{\frac{n}{2}}^k$, para $k \geq 0$.

Demonstração.

$$\begin{aligned} (\omega_n^k)^2 &= (\omega_n^{k+\frac{n}{2}})^2 \\ (\omega_n^k)^2 &= (\omega_n^k \cdot \omega_n^{\frac{n}{2}})^2 \\ (\omega_n^k)^2 &= (\omega_n^k \cdot (-1))^2 \text{ (Lema 2)} \\ (\omega_n^k)^2 &= (\omega_n^k)^2 \\ 1 &= 1 \end{aligned}$$

E pelo Lema 1 (do Cancelamento), temos que $(\omega_n^k)^2 = \omega_{\frac{n}{2}}^k$, para $k \geq 0$. □

Lema 4. (do Somatório). Para qualquer inteiro $n > 0$ e um inteiro não-zero k não múltiplo de n, $\sum_{j=0}^{n-1}(\omega_n^k)^j = 0$.

Demonstração. Pela soma da progessão geométrica, temos:

$$\begin{aligned}\sum_{j=0}^{n-1}(\omega_n^k)^j &= \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} \\ &= \frac{(e^{i2\pi\frac{k}{n}})^n - 1}{e^{i2\pi\frac{k}{n}} - 1} \\ &= \frac{e^{i2\pi k} - 1}{e^{i2\pi\frac{k}{n}} - 1} \\ &= \frac{1-1}{e^{i2\pi\frac{k}{n}} - 1} = 0\end{aligned}$$

O denominador nunca será igual zero, pois $e^{i2\pi\frac{k}{n}} = 1$ apenas quando k é múltiplo de n, o que é garantido pelo lema que não acontecerá. □

3.2.2 Transformada Discreta de Fourier

É possível perceber a partir do Lema 3 (da Metade), que as raízes ω_n^k e $\omega_n^{k+\frac{n}{2}}$ têm o mesmo quadrado, que é igual a $\omega_{\frac{n}{2}}^k$. Ou seja, no algoritmo de divisão e conquista podemos utilizar as n-ésimas raízes da unidade como os pontos de X a serem avaliados em num polinômio de grau $n-1$. Desta forma, a quantidade de elementos do conjunto diminuirá pela metade a cada nível da recursão.

Figura 3 – As n-ésimas raízes da unidade para n = 8, 4, 2 e 1

$$\begin{aligned}X &= \{-1, 1, -i, i, -\frac{\sqrt{2}}{2}(-1+i), \frac{\sqrt{2}}{2}(-1+i), -\frac{\sqrt{2}}{2}(1+i), \frac{\sqrt{2}}{2}(1+i)\} \\ X^2 &= \{-1, 1, i, -i\} \\ X^4 &= \{-1, 1\} \\ X^8 &= \{1\}\end{aligned}$$

O vetor das valorações de um polinômio $A(x)$, de grau $n-1$, nas n-ésimas raízes da unidade $\omega_n^0, \omega_n^1, ..., \omega_n^{n-1}$ é chamado de **Transformada Discreta de Fourier (DFT)**:

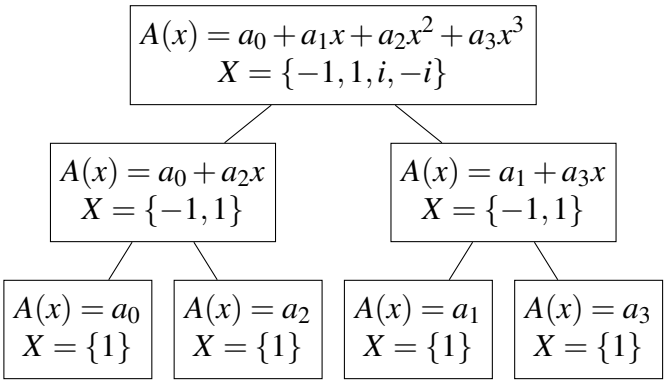
$$\begin{aligned}DFT_j(A) &= A(\omega_n^j) \\ &= \sum_{k=0}^{n-1} a_k \omega_n^{jk}\end{aligned}$$

Algoritmo 2: Implementação recursiva da Transformada Rápida de Fourier (FFT)

```
1 FFT(A)
2 n = A.tamanho                                   // Sempre é potência de 2

3 if n == 1 then
4     return (A_0, A_0, ..., A_0)                 // Vetor de tamanho n
5 X = (ω_n^0, ω_n^1, ..., ω_n^{n-1})

6 A_par = (A_0, A_2, ..., A_{n-2})
  A_impar = (A_1, A_3, ..., A_{n-1})

7 DFT_par = FFT(A_par)
  DFT_impar = FFT(A_impar)

8 return DFT_par + X * DFT_impar
```

Segue uma representação da árvore de recursão do algoritmo FFT para um exemplo em que temos um polinômio de grau 3 e queremos avaliá-lo nas 4 raízes da unidade ω_4^0, ω_4^1, ω_4^2 e ω_4^3:

Figura 4 – Árvore de recursão do algoritmo FFT para um polinômio de grau 3



Observemos, agora, que a cada chamada recursiva da função, o tamanho do vetor de coeficientes e o conjunto de ponto a serem avaliados nesta chamada diminui pela metade. Ou seja, para cada nó da árvore de recursão do algoritmo (de um total de $2n-1$), o polinômio referente a este nó terá que ser avaliado apenas em $\frac{n}{2^p}$ pontos, onde p é a profundidade do nó na árvore. Portanto, este algoritmo segue a seguinte recorrência:

$$T(n,|X|) = 2T(\frac{n}{2},|X^2|) + O(n+|X|)$$
$$T(n,|X|) = 2T(\frac{n}{2},\frac{|X|}{2}) + O(n+|X|)$$
$$T(n,|X|) = O(n \cdot \log_2 n)$$

3.2.3 Transformada Discreta Inversa de Fourier

Agora que sabemos converter um polinômio de sua forma de coeficientes para a forma de ponto-valor, nos resta fazer o caminho inverso: transformar o polinômio da forma de ponto-valor para a forma de coeficientes. Ou seja, realizar a operação inversa da Transformada Discreta de Fourier (DFT).

Para calcular a DFT tinhamos a seguinte relação:

$$\begin{bmatrix} (\omega_n^0)^0 & (\omega_n^0)^1 & (\omega_n^0)^2 & \cdots & (\omega_n^0)^{n-1} \\ (\omega_n^1)^0 & (\omega_n^1)^1 & (\omega_n^1)^2 & \cdots & (\omega_n^1)^{n-1} \\ (\omega_n^2)^0 & (\omega_n^2)^1 & (\omega_n^2)^2 & \cdots & (\omega_n^2)^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (\omega_n^{n-1})^0 & (\omega_n^{n-1})^1 & (\omega_n^{n-1})^2 & ... & (\omega_n^{n-1})^{n-1} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} DFT_0 \\ DFT_1 \\ DFT_2 \\ \vdots \\ DFT_{n-1} \end{bmatrix}$$

Logo, para obter a operação inversa da DFT, temos:

$$\begin{bmatrix} (\omega_n^0)^0 & (\omega_n^0)^1 & (\omega_n^0)^2 & \cdots & (\omega_n^0)^{n-1} \\ (\omega_n^1)^0 & (\omega_n^1)^1 & (\omega_n^1)^2 & \cdots & (\omega_n^1)^{n-1} \\ (\omega_n^2)^0 & (\omega_n^2)^1 & (\omega_n^2)^2 & \cdots & (\omega_n^2)^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (\omega_n^{n-1})^0 & (\omega_n^{n-1})^1 & (\omega_n^{n-1})^2 & ... & (\omega_n^{n-1})^{n-1} \end{bmatrix}^{-1} \cdot \begin{bmatrix} DFT_0 \\ DFT_1 \\ DFT_2 \\ \vdots \\ DFT_{n-1} \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

A partir do próximo lema que será apresentado, concluíremos que realizar operação inversa da DFT é um problema quase idêntico a calcular a própria DFT.

Lema 5. Para todo $0 \leq j,k < n$, o valor da matriz V_n^{-1} na posição (j,k) é $\frac{\omega_n^{-jk}}{n}$.

Demonstração. Mostraremos que $V_n^{-1} \cdot V_n = I_n$, onde I_n é a matriz identidade de dimensões n x n.

Denotamos W_{jl} o elemento de $V_n^{-1} \cdot V_n$ na posição (j,l).

$$\begin{aligned} W_{jl} &= \sum_{k=0}^{n-1} (\frac{\omega_n^{-kj}}{n}) \cdot (\omega_n^{kl}) \\ &= \sum_{k=0}^{n-1} \frac{\omega_n^{k(l-j)}}{n} \end{aligned}$$

Se $j = l$, então:

$$\begin{aligned} W_{jl} &= \sum_{k=0}^{n-1} \frac{\omega_n^0}{n} \\ &= \sum_{k=0}^{n-1} \frac{1}{n} = 1 \end{aligned} \tag{3.1}$$

Caso contrário, pelo Lema 4 (do Somatório), a soma é 0. É possível utilizar o lema pois $(l-j)$ não é divisível por n, já que $-(n-1) \leq (l-j) \leq (n-1)$. □

Logo, concluímos que para calcular a **Transformada Inversa Discreta de Fourier (IDFT)**:

$$a_j(DFT) = \sum_{k=0}^{n-1} \frac{DFT_k \cdot \omega_n^{-jk}}{n}$$

Segue, em pseudocódigo, a implementação do cálculo da inversa, que é quase idêntico ao código da DFT:

Algoritmo 3: Implementação recursiva da Transformada Rápida Inversa de Fourier (I_FFT)

```
1 I_FFT(DFT)
2 n = DFT.tamanho                                  // Sempre é potência de 2

3 if n == 1 then
4     return (DFT_0, DFT_0, ..., DFT_0)            // Vetor de tamanho n
5 X = (ω_n^0, ω_n^-1, ..., ω_n^-(n-1))

6 DFT_par = (DFT_0, DFT_2, ..., DFT_n-2)
  DFT_impar = (DFT_1, DFT_3, ..., DFT_n-1)

7 A_par = I_FFT(DFT_par)
  A_impar = I_FFT(DFT_impar)

8 return (A_par + X * A_impar)/n
```

Capítulo 4

CONVOLUÇÃO

Uma convolução entre duas funções de domínio discreto é uma operação que produz uma terceira função, que é um somatório do produto de duas funções, em que uma das funções é mantida da mesma forma e a outra é invertida e deslocada.

Formalmente, uma convolução entre duas funções de domínio discreto, onde f e g são sequências de tamanho n, é definida como (WIKIPEDIA, 2019):

$$(f * g)(j) = \sum_{k=0}^{j} f(k) \cdot g(j-k), \forall 0 \leq j \leq 2n-2$$

Se interpretarmos as funções f e g como se fossem coeficientes de dois polinômios, percebemos que a forma em que descrevemos a convolução é similar à forma em que descrevemos a multiplicação de polinômios no Capítulo 2. Portanto, podemos chegar à conclusão de que, se interpretarmos os valores das funções como coeficientes de polinômios, realizar a convolução entre duas funções de domínio discreto é equivalente a multiplicar dois polinômios.

Teorema 1. (da Convolução). A convolucão entre duas funções f e g, ambas com domínios de tamanho n, onde n é potência de 2, é:

$$f * g = DFT^{-1}(DFT(f) \cdot DFT(g))$$

Onde $\cdot$ é o produto ponto-a-ponto entre os vetores da DFT, que devem ter tamanho $2n$.

Algoritmo 4: Multiplicação de polinômios utilizando FFT

```
1 Multiplica-Polinomios(A, B)
2 DFT_A = FFT(A)
  DFT_B = FFT(B)

3 DFT_C = DFT_A · DFT_B
  C = I_FFT(DFT_C)

4 return C
```

A seguir, mostraremos exemplos de problemas que envolvem convoluções e que podem ser resolvidos utilizando a Transformada Rápida de Fourier (FFT).

4.1 Exemplos

4.1.1 Multiplicação de números grandes

Na computação é comum utilizarmos cadeia de caracteres para representar números grandes, aqueles que não podem ser representados pela arquitetura da máquina. Quando estamos trabalhando com este tipo de representação, é necessário implementar as operações básicas, como por exemplo a multiplicação.

Entretanto, utilizar o algoritmo trivial para multiplicar dois números tem complexidade $O(n^2)$ em tempo, onde n é a quantidade de dígitos dos números a serem multiplicados. Mostraremos agora uma solução utilizando o FFT.

Temos dois números A e B, representados como cadeia de caracteres, cada um contendo n dígitos:

$$S_a = a_{n-1}...a_2a_1a_0, 0 \leq a_j \leq 9$$
$$S_b = b_{n-1}...b_2b_1b_0, 0 \leq b_j \leq 9$$

Onde:

$$A = a_{n-1}10^{n-1} + ... + a_2 10^2 + a_1 10^1 + a_0$$
$$B = b_{n-1}10^{n-1} + ... + b_2 10^2 + b_1 10^1 + b_0$$

Desta forma, percebemos que é possível representar cada um dos números como um polinômio, onde o j-ésimo dígito do número é representado pelo j-ésimo coeficiente do polinômio:

$$P_A(x) = a_0 + a_1x^1 + a_2x^2 + ... + a_{n-1}x^{n-1}$$
$$P_B(x) = b_0 + b_1x^1 + b_2x^2 + ... + b_{n-1}x^{n-1}$$

Portanto, para obter $C = A \cdot B$, basta representarmos ambos os números A e B como

polinômios e efetuar a multiplicação destes através do FFT.

Algoritmo 5: Multiplicação de números grandes

```
1 Multiplica-Numeros(SA, SB)
2 SA = inverte(SA)
  SB = inverte(SB)

3 DFT_A = FFT(SA)
  DFT_B = FFT(SB)

4 DFT_C = DFT_A · DFT_B
  SC = I_FFT(DFT_C)

5 return C
```

A complexidade deste algoritmo é $O(n \cdot log_2 n)$.

4.1.2 Todas as somas entre duas listas de números

São dadas duas listas de números inteiros $A = [a_0, a_1, ..., a_{|A|-1}]$ e $B = [b_0, b_1, ..., b_{|B|-1}]$. O objetivo do problema é encontrar as somas $A[j] + B[k]$ e quantos pares geram esta soma, para todo $0 \leq j < |A|$ e $0 \leq k < |B|$ (CPALGORITHMS, 2019).

Criaremos dois polinômios, um para cada lista, onde o j-ésimo coeficiente de cada um deles é a quantidade de elementos j nas respectivas listas:

$$P_A(x) = Q_0(A)x^0 + Q_1(A)x^1 + ... + Q_{max(A)}(A)x^{max(A)}$$
$$P_B(x) = Q_0(B)x^0 + Q_1(B)x^1 + ... + Q_{max(B)}(B)x^{max(B)}$$

Onde $Q_j(L)$ é uma função que diz quantos elementos j existe na lista L e $max(L)$ é o maior valor que a lista L contém.

Ao multiplicarmos os dois polinômios, obteremos no j-ésimo coeficiente do polinômio resultante a quantidade de somas iguais a j entre os pares de valores de A e B.

Segue um exemplo, em que temos as seguintes listas A e B, e a frequência dos elementos em cada lista:

$$A = [0, 1, 1, 2, 3, 3, 3]$$
$$B = [2, 2, 2, 5]$$
$$Q_{0..3}(A) = [1, 2, 1, 3]$$
$$Q_{0..5}(B) = [0, 0, 3, 0, 0, 1]$$

E os dois polinômios P_A e P_B, criados a partir do número de ocorrências dos valores nas listas:

$$P_A(x) = 1x^0 + 2x^1 + 1x^2 + 3x^3$$
$$P_B(x) = 3x^2 + 1x^5$$

Multiplicando os polinômios, obtemos:

$$P_A(x)P_B(x) = (1x^0 + 2x^1 + 1x^2 + 3x^3)(3x^2 + 1x^5)$$
$$P_A(x)P_B(x) = 3x^2 + 6x^3 + 3x^4 + 10x^5 + 2x^6 + 1x^7 + 3x^8$$

Portanto, existem 3 pares de valores em que a soma resultam em 2, 6 que resultam em 3, 3 que resultam em 4, 10 que resultam em 5, 2 que resultam em 6, 1 que resulta em 7 e 3 que resultam em 8.

Segue, em pseudocódigo, o algoritmo que resolve este problema:

Algoritmo 6: Encontrar todas as somas entre pares de duas listas

1 `Encontra-Somas(A, B)`
2 $QA = conta\_ocorrencias(A)$
$QB = conta\_ocorrencias(B)$

3 $P_A = [QA_0, QA_1, ..., QA_{max(A)}]$
$P_B = [QB_0, QB_1, ..., QB_{max(B)}]$

4 $DFT_A = FFT(P_A)$
$DFT_B = FFT(P_B)$

5 $DFT_C = DFT_A \cdot DFT_B$
$C = I\_FFT(DFT_C)$

6 **return** C

A complexidade deste algoritmo é $O(n \cdot log_2 n)$.

Capítulo 5

CONCLUSÃO

A intenção deste artigo foi introduzir o leitor à Transformada Rápida de Fourier (FFT), mostrando principalmente como ela pode ser utilizada para multiplicar polinômios de forma eficiente e, por consequência, realizar convoluções em funções de domínio discreto. Espero que o artigo seja útil para estudantes de computação, em especial para os que participam da Maratona de Programação, visto que nela existem muitas aplicações para os conceitos aqui apresentados.

A inspiração para este trabalho veio primeiramente do GEMA (Grupo de Estudos para Maratona de Programação), um grupo de extensão do ICMC, no qual eu fui apresentado ao tópico pela primeira vez e ao qual eu sou muito grato por todo conhecimento obtido ao longo da graduação. Além do GEMA, algumas disciplinas que foram bastante úteis para a elaboração deste trabalho foram: *Introdução à Ciência de Computação I*, *Introdução à Ciência de Computação II*, *Algoritmos e Estruturas de Dados* e *Matemática Discreta I*, disciplinas que me forneceram uma base bastante sólida em fundamentos de lógica, programação e matemática, me ensinaram os principais algoritmos e estruturas de dados existentes e também a fazer análise de complexidade assintótica de algoritmos.

Algumas considerações e sugestões que eu posso fazer das disciplinas da graduação são: criação de uma disciplina obrigatória relacionada a Aprendizado de Máquina, pois é uma área bastante importante, a qual muitos alunos irão se deparar ao longo de suas carreiras; oferecimento de uma disciplina que envolva matemática combinacional e uma abordagem mais teórica de grafos, como já foi oferecida anteriormente com *Matemática Discreta II*; criação de uma disciplina dedicada exclusivamente para Álgebra Linear, e não juntamente a Equações Diferenciais Ordinárias como é atualmente, pois é uma disciplina importante em várias áreas da computação, como Aprendizado de Máquina; redução da carga horária de disciplinas relacionadas a engenharia de software, pois ela é bastante extensa, com *Análise e Projetos Orientados a Objetos*, *Engenharia de Software* e *Sistemas de Informação*, e o conteúdo é muitas vezes repetitivo.

Por fim, eu felizmente posso dizer que a formação que o curso de Ciências de Computação me ofereceu foi muito boa. A estrutura que o ICMC me forneceu foi excelente, tanto em relação à infraestrutura de salas de aulas e laboratórios, quanto em relação à professores que tive ao longo da graduação e aos funcionários do instituto.

REFERÊNCIAS

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms**. The MIT Press, 2009. ISBN 0262033844. Disponível em: <https://www.ebook.de/de/product/8474892/thomas_h_cormen_charles_e_leiserson_ronald_l_rivest_clifford_stein_introduction_to_algorithms.html>. Citado na página 23.

CPALGORITHMS. **Fast Fourier transform**. 2019. <https://cp-algorithms.com/algebra/fft.html>. [Online; accessed 05-June-2019]. Citado na página 31.

FRANCO, N. B. **Cálculo Numérico**. [S.l.]: Pearson, 2006. ISBN 8576050870. Citado 2 vezes nas páginas 15 e 18.

HEIDEMAN, M. T.; JOHNSON, D. H.; BURRUS, C. S. Gauss and the history of the fast fourier transform. **Archive for History of Exact Sciences**, Springer Nature, v. 34, n. 3, p. 265–277, 1985. Citado na página 13.

ROCKMORE, D. N. The FFT: an algorithm the whole family can use. **Computing in Science & Engineering**, Institute of Electrical and Electronics Engineers (IEEE), v. 2, n. 1, p. 60–64, 2000. Citado na página 13.

WIKIPEDIA. **Convolution theorem — Wikipedia, The Free Encyclopedia**. 2019. <http://en.wikipedia.org/w/index.php?title=Convolution%20theorem&oldid=894700435>. [Online; accessed 05-June-2019]. Citado na página 29.