**Complexidade De
Problemas Multilineares**

Daniel Nelson Panario Rodríguez

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA OBTENÇÃO DO GRAU DE
MESTRE EM
MATEMÁTICA APLICADA

Área de Concentração: Ciência da Computação
Orientador: Prof. Dr. Arnaldo Mandel

*Durante a elaboração deste trabalho,
o autor recebeu apoio financeiro da CAPES*

–São Paulo, Março de 1991–

Porque sufrías mientras nacía esta tesis.

Porque extraño tus esperas a mis llamados.

Por saber lo que era para vos y
por saber que vos sabías lo que vos eras y sos para mí.

Porque no viste el final y no importa.

Porque quiero y punto.

Porque te quiero.

a mi madre

Resumo

Na área de Complexidade Algébrica tem-se estudado com muita profundidade como calcular o produto de dois elementos de uma álgebra qualquer, os assim chamados problemas bilineares. Neste trabalho, estendemos o estudo ao produto de n elementos de uma álgebra, o que chamamos de problemas multilineares. Apresentamos uma nova forma de definir e de caracterizar os algoritmos algébricos, mostramos cotas inferiores do problema de calcular um conjunto de funções em $K[x_1, \dots, x_n]$, definimos algoritmos multilineares, damos um gerador de algoritmos multilineares quase-ótimos e estudamos com detalhe o produto de n polinômios numa variável. Esses métodos se mostraram adequados para o produto de polinômios em várias variáveis densos. Os polinômios em várias variáveis esparsos são muito importantes na prática, sendo nosso modelo ineficiente para tratá-los. Analisamos um algoritmo para calcular polinômios esparsos por interpolação, comparando sua complexidade pelo modelo RAM e pelo modelo de computação multiplicativo.

Abstract

The problem of computing the product of two elements of an algebra, the so called bilinear problems, has been much studied in Algebraic Complexity. In this work we extend this study to the product of n elements of an algebra, what we call multilinear problems. We present a new way to define and characterize algebraic algorithms, show lower bounds to compute a set of functions in $K[x_1, \dots, x_n]$, define multilinear algorithms, develop a generator of quasi-optimal multilinear algorithms and study in detail the product of n univariate polynomials. These methods can be used to compute the product of dense multivariate polynomials. In practice, sparse multivariate polynomials are very important, but our computational model is inefficient for treating. We analyze an algorithm for computing a sparse multivariate polynomial using interpolation, and compare its complexity by RAM and multiplicative model.

Agradecimentos

Aqui não dá prá ser original: então copiemos !!!

“Mal acredito que terminei este trabalho! Claro que não fiz sozinho! Tem muitos mais *culpados* na jogada, vamos a eles.”

Ao Arnaldo por me introduzir no assunto, por não apressar o rio, pela dedicação e porque sua mão esta presente nesta tese.

A Joos Heintz por papos muitos interessantes e esclarecedores a respeito da área e do estado atual do conhecimento.

A Lucia pelo carinho, por segurar as pontas nos momentos de desespero e pela correção tipográfica (não só do português!).

A Alair Pereira do Lago, Alberto Pardo, Alfredo Viola, Ivan Pan, Nelson Mugayar Kuhl e Walter Ferrer por discutirmos algum ponto desta tese.

À população da sala 257 por poupar meu tempo em consultar o Aurélio.

Ao Coelho pela frase do começo e por refletir o meu estado atual que é o estado de todo ser humano em fase final de tese (pelo menos no *MAC-IME-USP* !!!).

Aos meus amigos do planeta Terra por existirem.

A todos, muito obrigado !!!

Conteúdo

1	Teoria da Complexidade Algébrica	1
1.1	Introdução Histórica	1
1.2	Problemas Bilineares	2
1.3	Cotas Inferiores	6
1.4	Limitações da Área	7
2	Modelo de Computação	10
2.1	Definições Gerais	12
2.1.1	Computações e seus comprimentos	12
2.1.2	Algoritmos e suas complexidades.	13
2.2	Exemplos	17
2.2.1	Produto de 2 polinômios numa variável	17
2.2.2	Produto de 3 números complexos	19
2.2.3	Cômputo de uma função dada	20
2.3	Resultados Básicos	21
2.3.1	Medida intrínseca de complexidade	21
2.3.2	Proposições	22
2.3.3	Pode uma divisão ser sempre substituída por uma multiplicação?	27
2.3.4	Cotas Inferiores	29
3	Teoria Multilinear	32
3.1	Algoritmo e Complexidade Multilinear	34
3.2	Exemplos	37
3.3	Gerador de Algoritmos Multilineares	41

4	Produto de Polinômios	45
4.1	Produto de Polinômios Numa Variável	45
4.1.1	Produto de polinômios com mesmo grau	46
4.1.2	Produto de polinômios com graus supercrescentes . . .	56
4.1.3	Produto de polinômios com graus quaisquer	60
4.1.4	Problemas de implementação	65
4.2	Polinômios em Várias Variáveis	67
4.2.1	Polinômios em várias variáveis densos	69
4.2.2	Polinômios esparsos	72
4.2.2.1	Considerações iniciais	72
4.2.2.2	Algoritmo para interpolar polinômios esparsos	74
4.2.2.3	Resolução de Sistemas de Toeplitz	75
4.2.2.4	Raízes inteiras de um polinômio	78
4.2.2.5	Resolução de Sistemas de Vandermonde Transpostos .	81
	Bibliografia	85

Capítulo 1

Teoria da Complexidade Algébrica

Na teoria da complexidade algébrica estuda-se o seguinte problema :

Sejam $\{x_1, x_2, \dots, x_n\}$ indeterminadas sobre um corpo K , e $\{f_1, f_2, \dots, f_m\}$ um conjunto de funções racionais contidas em $K(x_1, \dots, x_n)$. Considerando os elementos de $K \cup \{x_1, \dots, x_n\}$ como dados, computar otimamente o conjunto $\{f_1, f_2, \dots, f_m\}$.

Obviamente devemos definir o que é computar funções e qual é a relação de ordem segundo a qual considera-se o ótimo. Neste capítulo vamos descrever os principais resultados da área, deixando a definição de nosso modelo de computação para o próximo.

1.1 Introdução Histórica

Em 1954, Alexander Ostrowski [39], analisando o problema de avaliar um polinômio em uma variável, questionou-se se com respeito ao número de operações aritméticas, o método de Horner é o melhor possível. Victor Ya Pan [40] demonstrou que o método de Horner é ótimo se os coeficientes do polinômio são indeterminadas sobre o corpo considerado, i.e., se o polinômio

é genérico. Se o polinômio é previamente fixado, em geral, a regra de Horner não é ótima.

Existem muitos problemas relacionados com a avaliação de polinômios : avaliação de polinômios específicos, polinômios difíceis de computar, avaliação de um polinômio em vários pontos diferentes, etc. Estes temas foram bastante estudados até fins dos anos 70 por um grande número de pesquisadores. Um bom resumo dos artigos publicados até 1975, encontra-se no livro de Borodin e Munro [7].

1.2 Problemas Bilineares

Dentre os problemas que pertencem à área da complexidade algébrica, existe uma classe muito estudada : a dos problemas bilineares. Um problema é bilinear quando as funções a serem computadas são bilineares.

Podemos definir os problemas bilineares como:

Dados espaços vetoriais U, V e W sobre um mesmo corpo K , com suas respectivas bases, e uma função bilinear $\phi : U \times V \rightarrow W$, qual é o custo de computar $\phi(x, y) \in W \quad \forall (x, y) \in U \times V$ nas bases prefixadas ?

Há muitos problemas que podem ser escritos nesta forma. Considere uma álgebra qualquer: a operação de produto na álgebra tem como resultado um conjunto de formas bilineares! Então, por exemplo: computar o produto de matrizes, polinômios, números complexos, extensões algébricas, etc, são problemas bilineares.

Exemplo : o produto em $Q(\sqrt{2})$ é definido por :

$$(x_1 + x_2\sqrt{2})(y_1 + y_2\sqrt{2}) = (x_1y_1 + 2x_2y_2) + (x_1y_2 + x_2y_1)\sqrt{2}$$

e podemos formular o problema da seguinte forma :

dados $(x_1, x_2), (y_1, y_2) \in Q^2$, computar $(x_1y_1 + 2x_2y_2, x_1y_2 + x_2y_1) \in Q^2$. $\square$

A atenção centrou-se nos problemas bilineares a partir de três artigos surpreendentes da década de 60.

1) O primeiro foi o dos matemáticos russos Karatsuba e Ofman [33], aprofundado pelo também soviético Toom [59], sobre o produto de inteiros grandes. O algoritmo escolar para o produto de dois números inteiros de n dígitos é da ordem de n^2 operações aritméticas. Usando uma estratégia do tipo divisão e conquista, Karatsuba e Ofman diminuíram o número de produtos e, apesar do aumento do número de adições, conseguiram um algoritmo da ordem de $n^{\log 3}$. O método pode ser generalizado dividindo os números em mais de duas partes, obtendo um algoritmo da ordem de $n^{1+\epsilon}$ com $\epsilon > 0$, mas inviável na prática.

Toom descreveu outros métodos para reduzir o expoente, e no ano de 1971, Schönhage e Strassen [48] desenvolveram um algoritmo assintoticamente rápido para este problema da ordem de $n \log n \log \log n$.

2) O segundo resultado importante tem seu maior interesse na prática, ao contrário do anterior. A transformada de Fourier sempre foi uma ferramenta útil no estudo de sistemas, mas devido à grande quantidade de cálculos que envolvem a transformada discreta, sua aplicabilidade não tinha acompanhado a teoria. No ano de 1965, Cooley e Tukey [12] publicaram um algoritmo que reduz os cálculos de $\mathcal{O}(n^2)$ para $\mathcal{O}(n \log n)$. A partir deste artigo, a transformada rápida de Fourier começou a ser aplicada computacionalmente numa variedade de áreas da ciência e engenharia, como por exemplo : acústica, processamento de imagens e sinais, projeto eficiente de algoritmos, etc. Winograd desenvolveu algoritmos práticos para a transformada de Fourier (Winograd [65] [66] [67]).

3) O mais surpreendente dos três problemas é o produto de matrizes. Podemos formular o problema de multiplicar matrizes $n \times n$, da seguinte forma:

Dadas as matrizes $n \times n$ $A = (a_{ij} : 1 \leq i, j \leq n)$, $B = (b_{ij} : 1 \leq i, j \leq n)$ com entradas no corpo K , computar seu produto $C = (c_{ij} : 1 \leq i, j \leq n)$ onde:

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

As indeterminadas do problema são os elementos das matrizes, e as funções a computar são as formas bilineares correspondentes aos coeficientes do produto das matrizes. O método dado pela definição do produto de matrizes utiliza n^3 produtos e $n^3 - n^2$ somas de elementos de K , tendo um total da ordem de n^3 operações. Strassen em 1969 [52] propôs um método para multiplicar matrizes de dimensões 2×2 , com 7 produtos (e não 8), mas tendo 18 somas (e não 4).

Algoritmo de Strassen : forme os seguintes produtos

$$\begin{aligned}P_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\P_2 &= (a_{21} + a_{22})b_{11} \\P_3 &= a_{11}(b_{12} - b_{22}) \\P_4 &= a_{22}(-b_{11} + b_{21}) \\P_5 &= (a_{11} + a_{12})b_{22} \\P_6 &= (-a_{11} + a_{21})(b_{11} + b_{12}) \\P_7 &= (a_{12} - a_{22})(b_{21} + b_{22})\end{aligned}$$

então os coeficientes da matriz produto são:

$$\begin{aligned}c_{11} &= P_1 + P_4 - P_5 + P_7 \\c_{12} &= P_3 + P_5 \\c_{21} &= P_2 + P_4 \\c_{22} &= P_1 - P_2 + P_3 + P_6\end{aligned}$$

A grande vantagem de usar o algoritmo de Strassen é que os elementos das matrizes podem também ser matrizes, pois as entradas são de um anel arbitrário (não comutativo). Como no caso do produto de inteiros grandes, o algoritmo usa uma estratégia do tipo divisão e conquista.

Analisemos a ordem deste algoritmo. Supondo $n = 2^k$ para algum k , temos que se $M(n)$ é o número de operações aritméticas totais para matrizes de dimensões $n \times n$, o algoritmo de Strassen tem uma complexidade dada por:

$$\begin{aligned}M(2) &= 7 \\M(2n) &= 7M(n) + 18n^2 \quad n \geq 2\end{aligned}$$

A solução desta recorrência é da ordem de $n^{\log 7}$ (Aho et alli [1], Lucchesi et alli [36]). Se n não fosse potência de 2, podemos preencher as matrizes com zeros até a próxima potência de dois, duplicando o custo, sem alterar a ordem do algoritmo. Temos, então, que a ordem deste algoritmo é $n^{\log 7}$ contra n^3 do algoritmo da definição, e portanto a partir de certo n o algoritmo de Strassen é melhor.

O artigo de Strassen é essencial no desenvolvimento da complexidade algébrica. Algumas consequências são :

1. A idéia de reduzir o número de produtos para melhorar o desempenho assintótico dos algoritmos é devido ao fato de, em alguns problemas bilineares, a complexidade do produto ser a mesma que o número total de operações aritméticas do problema. Nosso modelo de computação e a medida de custo, estão profundamente influenciados por esta constatação. Sua definição é de Winograd [61] [63], e às vezes é chamado de modelo de computação multiplicativa.
2. O produto de matrizes é usado em outros problemas, numéricos ou não, portanto uma melhora na ordem dos algoritmos que implementam o produto, pode implicar numa melhora nas ordens de uma classe de problemas. Alguns problemas numéricos que estão nesta classe são: inversa de uma matriz (Winograd [62]), cálculo de determinantes, decomposição de matrizes na forma LUP (Bunch e Hopcroft [11]), avaliação de n polinômios em n pontos diferentes (Borodin e Munro [7]), polinômio característico (Keller [34]), etc. Problemas não numéricos ligados ao produto de matrizes são: fecho transitivo de um grafo dirigido (Munro [38]) e reconhecimento de linguagens livres de contexto (Valiant [60]).
3. Outro corolário importante do artigo de Strassen foi o desenvolvimento de algoritmos rápidos para o produto de matrizes. No ano de 1978, Pan [41] melhorou a ordem do algoritmo de Strassen para $n^{2.79}$. A grande contribuição do artigo de Pan está no método usado em seu algoritmo, sendo suas idéias usadas para melhorar a cota deste problema nos trabalhos sucessivos. Alguns artigos posteriores ao de Pan que melhoraram a ordem do problema, são: Bini et alli [6], Pan [42],

Coppersmith [13], Coppersmith e Winograd [14], Schönhage [50] [51] e Strassen [56] [57]. Os avanços nesta área até o ano de 1984 estão no livro de Pan [43].

1.3 Cotas Inferiores

Durante a década de 70 foram desenvolvidos algoritmos para muitos problemas bilineares (produto de complexos, polinômios, quatérnios, etc). Uma vez que temos algoritmos para um problema, é natural nos perguntarmos se um algoritmo é ótimo. Para responder a esta pergunta, devemos estudar as propriedades intrínsecas do problema, independentemente dos algoritmos que dispomos para resolvê-lo. Em outras palavras: algoritmos dão cotas superiores do problema, e invariantes que medem a complexidade do problema dão cotas inferiores.

A área de complexidade algébrica é uma das poucas sub-áreas da computação onde tem sido possível provar cotas inferiores não triviais para complexidade, o que a torna muito importante do ponto de vista da teoria.

Alguns resultados importantes são:

Problema	Cota inferior	Referência
Complexos(C)	3	Winograd [63]
Matrizes(M_n)	$2n^2 - 1$	Brockett e Dobkin [9]
Polinômios(P_n)	$2n + 1$	Winograd [63]
Polinômios módulo pol. $K[t]/(f)$	$2n - m$ grau(f) = n m fatores primos	Winograd [63]
Quatérnios	8	Dobkin [22] De Groote [16] Howell-Lafon [29]

A unificação destes problemas como sendo a multiplicação em álgebras de dimensão finita, foi o passo natural. Esta formulação abstrata permite definir

cotas de complexidade em termos das propriedades da álgebra considerada. As primeiras tentativas neste sentido são de Strassen [55] e Fiducia-Zalcstein [24] (álgebras com divisão). No ano de 1981, Alder e Strassen [3] publicaram o primeiro resultado completamente geral: a complexidade do produto de uma álgebra associativa A está limitada inferiormente por $2 \dim(A) - \#M(A)$, onde $\#M(A)$ é o número de ideais maximais de A .

Heintz e Morgestern [28] estudaram as álgebras associativas que atingem a cota inferior. De Groote e Heintz [19] e Büchi e Clausen [10] dão resultados para álgebras comutativas; De Groote e Heintz [21] dão uma cota inferior para álgebras de Lie semisimples. Todos estes artigos usam fortemente os teoremas de estruturas de álgebras: Wedderburn para associativas e Artin para comutativas.

De Groote [15] [16] [17] [18] caracterizou o conjunto dos algoritmos bilineares ótimos para computar uma forma bilinear. O conjunto dos algoritmos bilineares ótimos para computar o produto na álgebra de Lie das matrizes 2×2 , $sl(2, k)$, é estudado por Mirwald [37], enquanto que Fellman [23] dá algoritmos ótimos para o produto em álgebras locais simplesmente geradas. Um excelente livro que resume todos estes trabalhos é o livro de De Groote [20].

Basicamente existem duas formas de demonstrar teoremas sobre cotas inferiores: substituição e grau. O método de substituição, baseado no conceito de independência algébrica, é utilizado em todos os casos anteriores e o único usado neste trabalho. Com ele, só podemos obter cotas inferiores lineares; já o método do grau (Strassen [55], Schönhage [49] e Schnorr [47]), nos permite ter cotas inferiores não lineares (mais precisamente logarítmicas).

1.4 Limitações da Área

Esta área é essencialmente teórica, sendo os seus problemas de natureza algébrica. Por outro lado os problemas são de interesse prático, querendo-se desenvolver algoritmos rápidos e eficientes. Apesar dos algoritmos considerados terem grande valor teórico, em termos práticos nem sempre são viáveis. Enumeramos, a seguir, algumas das limitações encontradas.

1. Tamanho dos programas. Este elemento é muito importante no caso do produto de matrizes. Em geral, os programas rápidos são extensos e mais sofisticados que aqueles que seguem a definição do problema considerado. Isto faz com que os programas efetuem mais operações e consumam mais memória.
2. O termo principal que define a ordem no algoritmo de Strassen é $4.7n^{\log 7}$. Winograd apresentou um algoritmo com 7 produtos e 15 somas (contra 18 de Strassen), reduzindo a constante para 4.54, mas com a mesma ordem de Strassen (Winograd [64]). Uma análise cuidadosa produz um algoritmo da ordem de $3.92 n^{\log 7}$ (Fischer [25]). Winograd [63] provou que no mínimo 7 produtos são necessários para matrizes 2×2 e Probert [45] mostrou que 15 somas no mínimo são requeridas por todo algoritmo que use 7 produtos, portanto o caso 2×2 está fechado. Podemos usar algoritmos rápidos para o produto de matrizes de dimensões grandes e a partir de um certo tamanho de matriz, usar o método da definição. De qualquer forma, as constantes do processo são grandes e os algoritmos rápidos melhoram o algoritmo da definição apenas para matrizes de tamanho muito grande (segundo Fischer [25] $n = 2^{10}$).
3. Erros numéricos. Estes algoritmos usam uma grande quantidade de operações, tendo problemas numéricos nos computadores atuais.
4. Controle das recursões. Deve-se ter em conta o trabalho de processar e controlar as recursões, podendo-se desconsiderar este custo dependendo da forma de acesso às matrizes por hardware.
5. Conseqüências do modelo de computação. O modelo multiplicativo desconsidera as somas e produtos por escalares, centrando o seu estudo nos produtos não escalares. Dependendo da aplicação, pode acontecer que nos interesse contar um produto escalar como não sendo escalar. Isto acontece quando num produto a ser calculado várias vezes, um fator pode ser pré-computado, evitando seu cálculo repetidas vezes. Também é importante para melhorar o tempo do processo, diminuir o número de somas sem afetar os produtos não escalares.

6. Espaço de memória. A maioria dos algoritmos citados são recursivos; deve-se ter em conta, então, a memória usada pelas chamadas recursivas. Além deste consumo de memória estrutural, em geral os algoritmos da área usam cálculos intermediários que são armazenados em variáveis internas de memória.

De qualquer forma apesar das limitações explicitadas, os algoritmos considerados fornecem cotas superiores dos problemas e podem ser adaptados para a prática através de heurísticas adequadas.

Capítulo 2

Modelo de Computação

Para medirmos a complexidade de problemas e algoritmos devemos previamente definir o modelo de computação. Nesse sentido devemos tornar preciso o que entendemos por algoritmo e qual é a medida para o custo que estamos considerando.

1) Analisemos as entradas de um algoritmo.

Sempre vamos ter um conjunto de constantes, que supomos formar um corpo. Além das constantes, outras entradas são necessárias que dependem do problema considerado. Por exemplo, se temos que computar o produto de dois polinômios quaisquer numa variável :

$$p(x) = \sum_{i=0}^n a_i x^i \quad e \quad q(x) = \sum_{i=0}^m b_i x^i$$

é natural considerar $a_0, \dots, a_n, b_0, \dots, b_m$ como indeterminadas do problema, mas se o problema é o produto de duas matrizes :

$$A = (a_{ij})_{m \times n} \quad e \quad B = (b_{ij})_{n \times p}$$

então as variáveis seriam : $a_{11}, \dots, a_{mn}, b_{11}, \dots, b_{np}$. Independentemente do fato de que as entradas dependem do problema, qualquer algoritmo simplesmente calcula elementos do corpo gerado pelas suas entradas.

Vamos chamar de base da computação, B , ao conjunto de dados de entrada de um algoritmo. Se o corpo das constantes é K e as indeterminadas são $\{x_1, \dots, x_n\}$ então, em geral, a base de computação vai ser :

$$B = K \cup \{x_1, \dots, x_n\}$$

Porém, pode acontecer que queiramos, por exemplo, x_1^2 como sendo um dado de entrada do algoritmo; é por isso que sempre vamos deixar a possibilidade de $K \cup \{x_1, \dots, x_n\} \subseteq B$. Sempre K será um corpo de característica zero.

2) Um segundo ponto a considerar é a respeito da complexidade multiplicativa. Como dito no capítulo 1, foi mostrado que para os problemas que nos interessam, as multiplicações não escalares são as que importam. Não vamos contar o produto por uma constante como sendo uma multiplicação. Podemos pensar na multiplicação por constante como implementada em hardware. Por outro lado, se multiplicar por uma certa constante for caro, é melhor tratá-la como variável. Como somas são mais baratas que multiplicações, também não serão contadas para efeito de complexidade.

3) Uma terceira observação pode ser feita a respeito do fluxo de controle do algoritmo (Strassen [53] [54]): *dado um algoritmo que usa instruções de desvio, existe um outro algoritmo "straight-line" (i.e., sem instruções de decisão), que executa as mesmas computações em um aberto (de Zariski) no domínio da entrada.*

Para provar a observação, vamos supor que as entradas pertencem a um corpo K qualquer. Em cada passo do algoritmo efetuam-se operações aritméticas (+, −, *, /) sobre os dados. Seja a entrada um ponto no espaço K^n , onde K é o espaço vetorial dos dados; podemos considerar as entradas como sendo restritas a algum aberto na topologia de Zariski de K^n (uma referência sobre topologia de Zariski é Fulton [26]).

Na execução do algoritmo, chegamos numa instrução de decisão (como estamos considerando algoritmos que executam contas, podemos supor que o desvio é feito sobre algum predicado aritmético). Necessariamente, pelo menos um dos desvios será escolhido por todos os pontos de algum aberto de K^n . Tomamos esse desvio. Continuamos assim, sempre tomando o desvio que seja válido para um aberto de K^n . Deste modo, o algoritmo tem um caminho que será percorrido toda vez que as entradas pertençam a um aberto dado, e portanto, os resultados são válidos para todo ponto deste aberto. Como os resultados computados são funções racionais das entradas, eles são válidos para todos os pontos do espaço, exceto possivelmente para alguma variedade algébrica.

Conclusão : se as entradas pertencem a um conjunto $X \subseteq K^n$, $\exists X'$ aberto, tal que em $X' \cap X$ podemos computar as funções racionais das entradas sem usar desvios.

2.1 Definições Gerais

2.1.1 Computações e seus comprimentos

Sejam K o corpo das constantes, $\{x_1, \dots, x_n\}$ indeterminadas sobre K e B uma base de computação tal que : $K \cup \{x_1, \dots, x_n\} \subseteq B \subseteq K(x_1, \dots, x_n)$.

Definição 2.1 Uma seqüência $\mathcal{C} := (h_1, \dots, h_l)$ em $K(x_1, \dots, x_n)$ é uma *computação sobre B* se $\forall i \leq m$ temos que :

i) $h_i \in B$ ou

ii) $\exists j, k < i$ e $o \in \{+, -, *, /\}$ tal que $h_i = h_j o h_k$; onde $h_k \neq 0$ se o é $/$

Na definição anterior consideramos o corpo K estendido pelas indeterminadas $\{x_1, \dots, x_n\}$, porém, também vai nos interessar trabalhar no anel $K[x_1, \dots, x_n]$. Neste caso não estará definida a operação de divisão.

Definição 2.2 Seja $\mathcal{C}$ uma seqüência sobre uma base de computação B . Vamos chamar de *subespaço de $K[x_1, \dots, x_n]$ computado por $\mathcal{C}$* ao espaço vetorial gerado por $\mathcal{C} \cup B$.

Para cada $T \subseteq K[x_1, \dots, x_n]$, $\mathcal{C}$ é uma *computação para T* se o espaço computado por $\mathcal{C}$ contém T .

Vamos definir o comprimento, ou custo, de uma computação. Uma possibilidade seria tomar o número de passos da computação como seu comprimento. Porém, tendo em vista o comentário do começo do capítulo, vamos considerar o custo de somar, subtrair, multiplicar por constantes, ou dividir por

constantes, como sendo zero. Para isso, distinguimos os produtos e divisões que devem ser tidos em conta na complexidade, chamando-os de essenciais. A partir do comprimento de uma seqüência, podemos definir o comprimento de um conjunto $T \subseteq K[x_1, \dots, x_n]$, como o mínimo dos comprimentos de seqüências que computam T . Então, temos as seguintes definições :

Definição 2.3 Seja $C = (h_1, \dots, h_l)$ uma computação. Uma multiplicação $h_i = h_j * h_k$ para $j, k < i$ é chamada *escalar*, se h_j ou $h_k \in K$. Caso contrário é chamada *essencial*.

Uma divisão $h_i = h_j / h_k$ é chamada *escalar*, se $h_k \in K$. Caso contrário é chamada *essencial*.

Se C é um subconjunto de um espaço vetorial, denotaremos por $\langle C \rangle$ o subespaço gerado por C .

Definição 2.4 O comprimento $D(C)$ de uma computação C é o número de multiplicações e divisões essenciais de C . Para cada $T \subseteq K[x_1, \dots, x_n]$, definimos

$$D(T) := \min \{ D(C) : T \subseteq \langle C \rangle \}$$

Uma computação C de comprimento mínimo para T é chamada *ótima*.

2.1.2 Algoritmos e suas complexidades.

Nesta seção vamos dar uma definição geral de algoritmo algébrico que nos permitirá obter uma definição de complexidade elegante e independente. A idéia da definição é considerar os passos do algoritmo como sendo ou básicos, ou essenciais. Os básicos são combinações lineares das indeterminadas, passos anteriores e constantes. Os passos essenciais são produtos de dois passos anteriores, que não sejam constantes.

Definição 2.5 Um *algoritmo algébrico* $\mathcal{A}$ em $K[x_1, \dots, x_n]$ sobre uma base B , é uma m -upla $(p_1, \dots, p_m)$ tal que para cada j , $1 \leq j \leq m$:

- i) ou $p_j \in B$
- ii) ou $p_j \in K^{j-1}$
- iii) ou $p_j = (h, i)$ onde $h, i < j$

O número de elementos do tipo iii) é a complexidade do algoritmo $\mathcal{A}$, denotada por $D(\mathcal{A})$.

Observações :

1) Para cada passo p_j do algoritmo temos um polinômio p'_j associado. Os passos do algoritmo do tipo i) são chamados *básicos* e $p_j = p'_j$. Os passos do tipo ii) são chamados *somas*, e são tais que se $p_j = (a_1, \dots, a_{j-1})$ então representam :

$$p'_j(x_1, \dots, x_n) = \sum_{l=1}^{j-1} a_l * p'_l(x_1, \dots, x_n)$$

Os passos da forma $p_j = (h, i)$ tal que $h, i < j$, são chamados *multiplicações* e representam :

$$p'_j(x_1, \dots, x_n) = p'_h(x_1, \dots, x_n) * p'_i(x_1, \dots, x_n)$$

Por ser mais ilustrativo, vamos freqüentemente descrever os passos através dos seus polinômios.

2) Uma consequência importante do modelo escolhido é que o custo de computar uma função não depende da base considerada, pois mudança de base tem custo zero. Deve-se ter claro que, dado um algoritmo para computar um problema algébrico, os elementos estão considerados numa base prefixada. Com respeito a essa base, a mudança de base tem custo zero, e então o algoritmo independe de base.

3) Um outro ponto que queremos destacar é a diferença entre função a computar e as fórmulas que usamos para isso. A idéia é tentar mudar de fórmula, de modo tal que obtenhamos a mesma função com menos produtos.

4) As vezes vai nos interessar que um algoritmo algébrico tenha a possibilidade de dividir. Quando for o caso, teremos passos chamados *divisões* da forma $p_j = h/i$ onde $h, i < j$, que representam:

$$p'_j(x_1, \dots, x_n) = p'_h(x_1, \dots, x_n) / p'_i(x_1, \dots, x_n)$$

5) Nos exemplos de algoritmos, vamos dar duas versões; à direita na forma que segue da definição, e à esquerda numa outra maneira mais informal segundo a convenção :

i) os primeiros $n + 1$ passos do algoritmo são : $1, x_1, \dots, x_n$

ii) os passos seguintes serem notados como :

$$p_{n+1+r} = \begin{cases} B_r & \text{se } p_{n+1+r} \text{ é básico} \\ S_r & \text{se } p_{n+1+r} \text{ é uma soma} \\ M_r & \text{se } p_{n+1+r} \text{ é uma multiplicação} \end{cases}$$

6) A inclusão das constantes é necessária para ter uma definição geral de algoritmo algébrico (observe que basta ingressar o número 1 para ter todas as constantes, pois 1 é um gerador do corpo K).

Se queremos computar a função :

$$f(x, y, z) = xyz + 2xz = (xz) * (y + 2)$$

alguns dos algoritmos que permitem computar f são :

a)

1	$p_1 = 1$	$p'_1 = 1$
x	$p_2 = x$	$p'_2 = x$
y	$p_3 = y$	$p'_3 = y$
z	$p_4 = z$	$p'_4 = z$
$M_1 = x * y$	$p_5 = (2, 3)$	$p'_5 = p'_2 \cdot p'_3 = xy$
$M_2 = x * z$	$p_6 = (2, 4)$	$p'_6 = p'_2 \cdot p'_4 = xz$
$M_3 = M_1 * z$	$p_7 = (5, 4)$	$p'_7 = p'_5 \cdot p'_4 = xyz$
$f = 2M_2 + M_3$	$p_8 = (0, 0, 0, 0, 0, 2, 1)$	$p'_8 = 2p'_6 + p'_7 = xyz + 2xz$

b)

1	$p_1 = 1$	$p'_1 = 1$
x	$p_2 = x$	$p'_2 = x$
y	$p_3 = y$	$p'_3 = y$
z	$p_4 = z$	$p'_4 = z$
$M_1 = x * z$	$p_5 = (2, 4)$	$p'_5 = p'_2 \cdot p'_4 = xz$
$S_2 = y + 2$	$p_6 = (2, 0, 1, 0, 0)$	$p'_6 = 2p'_1 + p'_3 = 2 + y$
$f = M_1 * S_2$	$p_7 = (5, 6)$	$p'_7 = p'_5 \cdot p'_6 = (xz) \cdot (2 + y)$

No algoritmo a), não se usam constantes (observe que a única soma é p_8 na qual temos 0 para o termo constante), e a sua complexidade é 3 (os passos cujos nomes começam com 'M'), enquanto que no algoritmo b) usam-se constantes e tem complexidade 2. Portanto, se não consideramos constantes estamos perdendo algoritmos (que talvez sejam ótimos).

Se F fosse um conjunto de formas multilineares em $K[x_{11}, \dots, x_{nm_n}]$ a necessidade é a mesma. Seja, por exemplo, $f(x, y) = xy$ em $K[x, y]$. Podemos calcular f através do algoritmo que vem da própria fórmula, que deixamos a cargo do leitor, ou através do seguinte algoritmo :

1	$p_1 = 1$	$p'_1 = 1$
x	$p_2 = x$	$p'_2 = x$
y	$p_3 = y$	$p'_3 = y$
$S_1 = 1 + x$	$p_4 = (1, 1, 0)$	$p'_4 = p'_1 + p'_2 = 1 + x$
$S_2 = 1 + y$	$p_5 = (1, 0, 1, 0)$	$p'_5 = p'_1 + p'_3 = 1 + y$
$M_3 = S_1 * S_2$	$p_6 = (4, 5)$	$p'_6 = p'_4 * p'_5 = (1 + x)(1 + y)$
$f = M_3 - x - y - 1$	$p_7 = (-1, -1, -1, 0, 0, 1)$	$p'_7 = -p'_1 - p'_2 - p'_3 + p'_6 = xy$

Analogamente às definições 2.2 e 2.4, temos para algoritmos as definições de K -subespaço gerado e algoritmo ótimo para um subconjunto qualquer de $K[x_1, \dots, x_n]$, para as quais usaremos as mesmas notações.

Definição 2.6 Seja $\langle \mathcal{A} \rangle$ o subespaço de $K[x_1, \dots, x_n]$, gerado pelo algoritmo algébrico $\mathcal{A} := (p_1, \dots, p_m)$ sobre a base de computação B . Para cada $T \subseteq K[x_1, \dots, x_n]$, $\mathcal{A}$ é um *algoritmo algébrico* para T , se $T \subseteq \langle \mathcal{A} \rangle$.

$\mathcal{A}$ é um *algoritmo algébrico ótimo* para T se :

$$D(\mathcal{A}) = \min \{ D(\mathcal{A}') : T \subseteq \langle \mathcal{A}' \rangle \}$$

e dizemos que $D(\mathcal{A})$ é a *complexidade* de T , denotada por $D(T)$.

Pela definição 2.5 a complexidade de um algoritmo $\mathcal{A}$ é o número de elementos do tipo iii) em $\mathcal{A}$, isto é, o número de multiplicações de $\mathcal{A}$. Como os fatores dos produtos podem ser constantes, as multiplicações podem não ser essenciais. Neste caso, existe um outro algoritmo $\mathcal{A}'$ tal que: $\langle \mathcal{A} \rangle = \langle \mathcal{A}' \rangle$ mas $D(\mathcal{A}') \leq D(\mathcal{A})$.

2.2 Exemplos

2.2.1 Produto de 2 polinômios numa variável

Sejam dois polinômios quaisquer sobre um corpo K :

$$p(x) = \sum_{i=0}^n a_i x^i \quad e \quad q(x) = \sum_{i=0}^m b_i x^i$$

Seu produto é:

$$p(x)q(x) = \sum_{i=0}^{n+m} c_i x^i \quad \text{onde} \quad c_i = \sum_{j=0}^i a_j b_{i-j}$$

O problema é calcular os c_i sobre a base $\{a_i\} \cup \{b_i\}$.

O algoritmo que damos a seguir baseia-se no esquema de interpolação de Lagrange e está em Winograd [67]. Lembramos que estamos supondo que $\text{car}(K) = 0$ (bastaria apenas a existência de $n+m+1$ constantes diferentes).

Sejam $g_0, g_1, \dots, g_{n+m}$ constantes diferentes. Os passos do algoritmo são:

i) Calcule: $r(g_i) := p(g_i) q(g_i) \quad i = 0, \dots, n+m$

ii) Considere:

$$L_i(x) := \frac{\prod_{j \neq i} (x - g_j)}{\prod_{j \neq i} (g_i - g_j)} = \sum_{k=0}^{n+m} r_{ik} x^k \quad i = 0, \dots, n+m$$

iii) Finalmente :

$$c_k = \sum_{i=0}^{n+m} r(g_i) r_{ik} \quad k = 0, \dots, n+m$$

Vamos provar que o algoritmo calcula os coeficientes do produto dos polinômios:

$$\begin{aligned} p(x)q(x) &= \sum_{i=0}^{n+m} p(g_i)q(g_i) \frac{\prod_{j \neq i} (x - g_j)}{\prod_{j \neq i} (g_i - g_j)} \\ &= \sum_{i=0}^{n+m} r(g_i) \left(\sum_{k=0}^{n+m} r_{ik} x^k \right) \\ &= \sum_{i=0}^{n+m} \left(\sum_{k=0}^{n+m} r(g_i) r_{ik} \right) x^k \\ &= \sum_{k=0}^{n+m} \left(\sum_{i=0}^{n+m} r(g_i) r_{ik} \right) x^k \end{aligned}$$

Logo:

$$p(x)q(x) = \sum_{k=0}^{n+m} c_k x^k$$

Analisemos a complexidade do algoritmo. Calcular $p(g_i)$ e $q(g_i)$ tem custo zero, pois $g_i \in K$. Para cada i , o custo de computar $r(g_i)$ é 1. Pela própria definição de $L_i(x)$, seus coeficientes são elementos de K , já que as constantes $g_0, g_1, \dots, g_{n+m}$ foram determinadas a priori. Portanto podemos calcular cada coeficiente do produto com $n + m + 1$ multiplicações. Como todos os coeficientes usam as mesmas $n + m + 1$ multiplicações, o algoritmo de Winograd tem complexidade $n + m + 1$.

2.2.2 Produto de 3 números complexos

O produto dos três números complexos : $x_1 + ix_2, y_1 + iy_2, z_1 + iz_2$ é definido como sendo :

$$(x_1y_1z_1 - x_1y_2z_2 - x_2y_1z_2 - x_2y_2z_1) + i(x_1y_1z_2 + x_1y_2z_1 + x_2y_1z_1 - x_2y_2z_2)$$

O algoritmo abaixo implementa o produto de três números complexos com complexidade 6. Nele consideramos a seguinte convenção a fim de obter a mesma nomenclatura das definições :

- i) $x_3 := y_1, x_4 := y_2, x_5 := z_1, x_6 := z_2$.
- ii) à esquerda temos o algoritmo formulado informalmente, enquanto que na coluna da direita temos sua definição em termos formais.
- iii) os passos do algoritmo que correspondem a uma soma, foram rotulados com a letra 'S', os passos correspondentes a uma multiplicação com 'M', as funções intermediárias com ϕ e as duas componentes do produto com f_i . Já no algoritmo formal, todos os passos foram rotulados com a letra 'p'.

1		p_1	=	1	
x_1		p_2	=	x_1	
x_2		p_3	=	x_2	
y_1		p_4	=	y_1	
y_2		p_5	=	y_2	
z_1		p_6	=	z_1	
z_2		p_7	=	z_2	
M_1	:=	$x_1 * y_1$	p_8	=	(2, 4)
M_2	:=	$x_2 * y_2$	p_9	=	(3, 5)
S_3	:=	$x_1 + x_2$	p_{10}	=	(0, 1, 1, 0, 0, 0, 0, 0, 0)
S_4	:=	$y_1 + y_2$	p_{11}	=	(0, 0, 0, 1, 1, 0, 0, 0, 0, 0)
M_5	:=	$S_3 * S_4$	p_{12}	=	(10, 11)
ϕ_6	:=	$M_1 - M_2$	p_{13}	=	(0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0)
ϕ_7	:=	$M_5 - M_1 - M_2$	p_{14}	=	(0, 0, 0, 0, 0, 0, 0, 0, -1, -1, 0, 0, 1, 0)
M_8	:=	$\phi_6 * z_1$	p_{15}	=	(13, 6)
M_9	:=	$\phi_7 * z_2$	p_{16}	=	(14, 7)
S_{10}	:=	$\phi_6 + \phi_7$	p_{17}	=	(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0)
S_{11}	:=	$z_1 + z_2$	p_{18}	=	(0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0)
M_{12}	:=	$S_{10} * S_{11}$	p_{19}	=	(17, 18)
f_1	:=	$M_8 - M_9$	p_{20}	=	(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0)
f_2	:=	$M_{12} - M_8 - M_9$	p_{21}	=	(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1, 0, 0, 1, 0)

Os produtos essenciais no algoritmo, estão nos passos : 5, 6, 9, 14, 15 e 18.
Portanto a complexidade do algoritmo é 6. $\square$

2.2.3 Cômputo de uma função dada

Vamos supor que temos que computar a função :

$$f := x_1 y_1 z_1 w_1 + x_1 y_1 z_2 w_2 + x_2 y_2 z_1 w_1 + x_2 y_2 z_2 w_2$$

Uma forma de calcular a função f é através do algoritmo que segue da própria fórmula. A complexidade, aplicando esse algoritmo trivial, seria de 12 (cada termo da expressão tem um custo de 3, e são 4 termos). Esse algoritmo não é muito eficiente e rapidamente pode-se perceber possíveis melhoras. Fatorando, temos uma outra fórmula para f dada por:

$$(x_1 y_1 + x_2 y_2) * (z_1 w_1 + z_2 w_2)$$

O algoritmo que executa primeiro as operações dentro dos parênteses e depois multiplica os parênteses, tem complexidade 5 e calcula f . Apresentamos a

seguir este algoritmo :

1		p_1	=	1	
x_1		p_2	=	x_1	
x_2		p_3	=	x_2	
y_1		p_4	=	y_1	
y_2		p_5	=	y_2	
z_1		p_6	=	z_1	
z_2		p_7	=	z_2	
w_1		p_8	=	w_1	
w_2		p_9	=	w_2	
M_1	:=	$x_1 * y_1$	p_{10}	=	(2, 4)
M_2	:=	$x_2 * y_2$	p_{11}	=	(3, 5)
M_3	:=	$z_1 * w_1$	p_{12}	=	(6, 8)
M_4	:=	$z_2 * w_2$	p_{13}	=	(7, 9)
S_5	:=	$M_1 + M_2$	p_{14}	=	(0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0)
S_6	:=	$M_3 + M_4$	p_{15}	=	(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0)
f	:=	$S_5 * S_6$	p_{16}	=	(14, 15)

A complexidade do algoritmo é 5, e seus passos multiplicativos são : 10, 11, 12, 13 e 16. $\square$

2.3 Resultados Básicos

2.3.1 Medida intrínseca de complexidade

Nesta seção vamos ver que a definição 2.6 é independente do algoritmo considerado. Lembramos que se B é um subconjunto de espaço vetorial, denotaremos por $\langle B \rangle$ o subespaço gerado por B .

Definição 2.7 Sejam U e V dois K -subespaços vetoriais do anel dos polinômios $K[x_1, \dots, x_n]$. Dizemos que V cobre U se existem $p, q \in V$ com $p.q \notin V$ tal que:

$$U = V \oplus \langle p.q \rangle$$

Definição 2.8 Sejam $W_0, W_1, \dots, W_l$, K -subespaços vetoriais do anel dos polinômios $K[x_1, \dots, x_n]$. Se W_{i-1} cobre $W_i \ \forall i, i = 1, \dots, l$, dizemos que os subespaços formam uma *cadeia de cobertura*, denotada por:

$$W_0 \leq W_1 \leq W_2 \cdots \leq W_l$$

Seja $\mathcal{A} := (p_1, \dots, p_m)$ um algoritmo sobre B , D_A sua complexidade e os produtos essenciais: $\{h_1.f_1, \dots, h_{D_A}.f_{D_A}\}$, definimos :

$$\begin{aligned} W_0 &= L_K \langle B \rangle \\ W_i &= W_{i-1} \oplus \langle h_i.f_i \rangle \quad i = 1, \dots, D_A \end{aligned}$$

Como $h_i, f_i \in W_{i-1}$ mas $h_i.f_i \notin W_{i-1}$, temos que : W_{i-1} cobre $W_i \ \forall i$. A recorrência anterior define uma cadeia de cobertura:

$$W_0 \leq W_1 \leq W_2 \cdots \leq W_{D_A}$$

supondo sem perda de generalidade que para $i \neq j$, $\langle h_i.f_i \rangle \neq \langle h_j.f_j \rangle$.

Se $\mathcal{A}$ é um algoritmo para T , então $T \subseteq W_{D_A}$.

Como $\mathcal{A}$ é um algoritmo qualquer para T , pela definição de $D(T)$:

$$D(T) = \min \{l : W_0 \leq W_1 \leq W_2 \cdots \leq W_l \supseteq T\}$$

mostrando que a definição de complexidade é independente do algoritmo. $\square$

2.3.2 Proposições

Vamos enunciar algumas proposições básicas. A primeira delas, nos assegura a existência de algoritmos ótimos para computar subconjuntos de $K[x_1, \dots, x_n]$. A segunda indica que só temos complexidade nula quando

fazemos combinações lineares da base de computação. A terceira é óbvia : se um conjunto está incluído em outro, o custo de computar o primeiro é menor que o custo de computar o segundo. A quarta proposição é uma variação da segunda : computar combinações lineares da base de computação, não aumenta a complexidade.

Por indução no número de passos de um algoritmo, pode ser provado que todos os passos de um algoritmo são combinações lineares de suas multiplicações essenciais (proposição 2.5).

Proposição 2.1 *Se $T \subseteq K[x_1, \dots, x_n]$ é finito, existe um algoritmo $\mathcal{A} := (p_1, \dots, p_m)$, ótimo para T , tal que os elementos de T aparecem como elementos de $\{p_1, \dots, p_m\}$.*

Proposição 2.2 *Sejam $T = \{q_1, \dots, q_t\} \subseteq K[x_1, \dots, x_n]$ e B uma base de computação. Então :*

$$D(T) = 0 \Leftrightarrow q_i \in \langle B \rangle \quad \forall i = 1, 2, \dots, t$$

Proposição 2.3 *Sejam $T, T' \subseteq K[x_1, \dots, x_n]$ tal que $T \subseteq T'$. Então :*

$$D(T) \leq D(T')$$

Proposição 2.4 *Sejam $T, T' \subseteq K[x_1, \dots, x_n]$ com $T = \{q_1, \dots, q_t\}$ e $T' = \{q'_1, \dots, q'_t\}$, tal que : $q'_i = q_i + l_i$ e $l_i \in \langle B \rangle \quad \forall i = 1, 2, \dots, t$. Então:*

$$D(T) = D(T')$$

Proposição 2.5 *Sejam $\mathcal{A} := (p_1, \dots, p_m)$ um algoritmo e $(h_1, \dots, h_l)$ as suas multiplicações essenciais. Então $p_i \in \langle h_1, \dots, h_l \rangle \quad \forall i = 1, 2, \dots, m$.*

Se um algoritmo vai ser usado repetidas vezes, é bom ter um controle do crescimento dos dados de entrada. Dado um algoritmo para computar polinômios de grau $\leq g$ que usa fatores de grau maior que g , queremos construir um outro algoritmo que calcule o conjunto de funções mas que não use fatores de grau maior que g . O teorema abaixo nos dá uma forma de construir tal algoritmo.

Teorema 2.1 *Seja F um conjunto de funções em $K[x_1, \dots, x_n]$ com graus $\leq g$ e B uma base de computação. Existe um algoritmo $\mathcal{A}$ que computa F sobre B , tal que o polinômio associado a cada passo de $\mathcal{A}$ tem grau menor que g , e:*

$$D(F) \leq D(\mathcal{A}) \leq (g-1) D(F)$$

Demonstração .- Como o algoritmo $\mathcal{A}$ computa F , temos que:

$$D(F) \leq D(\mathcal{A})$$

Visando demonstrar a segunda desigualdade, seja $A' = (p'_1, \dots, p'_m)$ um algoritmo ótimo para F . Vamos construir o algoritmo desejado a partir de A' . Seja $L_i : i = 0, \dots, g$, o operador linear sobre o espaço dos polinômios que dá o termo homogêneo de grau i do polinômio e seja L o operador que dado um polinômio nos dá os termos de grau $\leq g$ do polinômio, i.e., $L = L_0 + \dots + L_g$. Lembramos que para simplificar trabalharemos com os polinômios como sendo os passos do algoritmo.

Vamos modificar A' de modo que:

$$\text{i) } \forall p' \in A' \quad \exists p \in \mathcal{A} \quad \text{tal que} \quad L(p') = L(p)$$

$$\text{ii) } L(p) = p \quad \forall p \in \mathcal{A}$$

A primeira condição nos diz que para cada passo do algoritmo A' , temos um passo em $\mathcal{A}$ que tem os mesmos termos até o grau g , e a segunda condição assegura-nos que todos os passos de $\mathcal{A}$ têm graus menores ou iguais que g .

Das duas condições concluímos que:

$$L(p') = L(p) = p$$

ou seja, que os passos de $\mathcal{A}$ são os de A' , considerando até o grau g .
A demonstração será feita por indução no número do passo.

Base: pela quinta observação da definição 2.5 de algoritmo algébrico, os primeiros passos do algoritmo pertencem a B , portanto: $p_1 = L(p'_1) = p'_1$

Passo indutivo: vamos supor que $p'_1, \dots, p'_{h-1}$ já foram modificados para $p_1, \dots, p_{h-1}$. Então para p'_h temos as seguintes possibilidades:

i) $p'_h \in B$ então $p_h = L(p'_h) = p'_h$

ii) $p'_h = \sum_{i=1}^{h-1} a_i * p'_i$ então:

$$p_h = L(p'_h) = L\left(\sum_{i=1}^{h-1} a_i * p'_i\right) = \sum_{i=1}^{h-1} a_i * L(p'_i) = \sum_{i=1}^{h-1} a_i * p_i$$

iii) $p'_h = (i, j)$ $i, j < h$, então $p_h = L(p'_h)$

Vamos substituir $L(p'_h)$ por uma seqüência de passos que computam $L_0(p'_h)$, $L_1(p'_h), \dots, L_g(p'_h)$. Operando:

$$\begin{aligned} L_0(p'_h) &= L_0(p_i) * L_0(p_j) \\ L_1(p'_h) &= L_0(p_i) * L_1(p_j) + L_1(p_i) * L_0(p_j) \\ L_2(p'_h) &= L_0(p_i) * L_2(p_j) + L_1(p_i) * L_1(p_j) + L_2(p_i) * L_0(p_j) \\ &\vdots \\ &\vdots \\ L_g(p'_h) &= L_0(p_i) * L_g(p_j) + L_1(p_i) * L_{g-1}(p_j) + \dots \\ &\quad \dots + L_{g-1}(p_i) * L_1(p_j) + L_g(p_i) * L_0(p_j) \end{aligned}$$

Uma primeira observação é que $L_0(p_i)$ e $L_0(p_j)$ são constantes, portanto as multiplicações efetivas são as $g(g-1)/2$ seguintes:

$$L_1(p_i) * L_1(p_j), L_1(p_i) * L_2(p_j), \dots, L_1(p_i) * L_{g-1}(p_j), L_2(p_i) * L_1(p_j), \dots, \\ L_2(p_i) * L_{g-2}(p_j), \dots, L_{g-2}(p_i) * L_1(p_j), L_{g-2}(p_i) * L_2(p_j), L_{g-1}(p_i) * L_1(p_j)$$

Como estamos interessados em $L = L_0 + L_1 + \dots + L_g$, fatorando a expressão, temos que:

$$p_h = L(p'_h) = L_1(p_i) * \sum_{l=1}^{g-1} L_l(p_j) + L_2(p_i) * \sum_{l=1}^{g-2} L_l(p_j) + \dots + L_{g-1}(p_i) * L_1(p_j)$$

Portanto podemos obter p_h a partir de p'_h com $g-1$ produtos, de onde:

$$\boxed{D(F) \leq D(\mathcal{A}) \leq (g-1) D(T)}$$

Observação: como a idéia do algoritmo é considerar os polinômios associados aos passos, truncados no grau g , seria mais natural tomar módulo I em cada passo, sendo I o ideal gerado por

$$\{x_{i_1} x_{i_2} \dots x_{i_g} : 1 \leq i_1 \leq i_2 \leq \dots \leq i_g \leq n\}$$

Porém, módulo I não é uma operação do nosso modelo de computação, e não pode ser, pois trabalhamos com formas polinomiais e não com polinômios. Se acrescentássemos essa operação com custo zero (é só desconsiderar os termos de grau maior que g), teríamos que o algoritmo seria ótimo, pois poderíamos substituir cada produto do algoritmo original por um produto aplicando a seguinte regra:

$$\begin{aligned} p_h &= p'_h \text{ mod } I \\ &= (p'_i * p'_j) \text{ mod } I \\ &= ((p'_i \text{ mod } I) * (p'_j \text{ mod } I)) \text{ mod } I \\ &= (p_i * p_j) \text{ mod } I \end{aligned}$$

2.3.3 Pode uma divisão ser sempre substituída por uma multiplicação?

Nesta seção mostraremos que para uma classe específica de problemas algébricos, algoritmos ótimos não necessitam de divisões. Porém, o uso de divisões pode diminuir a complexidade em casos particulares. Por exemplo, se temos que computar $f(x) = x^{31}$ sem usar divisões, temos um algoritmo com 8 produtos; usando divisões, temos um algoritmo ótimo com complexidade 6 (De Groote [20]).

Daremos também um algoritmo que substitui uma divisão por vários produtos e somas, para formas polinomiais quaisquer em várias variáveis. A idéia fundamental é substituir cada divisão por uma série de potências. Para isso, se q é um polinômio em $K[x_1, \dots, x_n]$ sem termo constante, temos que:

$$\frac{1}{1 - q} = \sum_{i=0}^{\infty} q^i$$

Como os resultados das operações são formas polinomiais, não necessitamos considerar a série de potências completa, senão até algum grau g . Assim, podemos substituir uma divisão por um número *finito* de somas e produtos. Devemos assegurar-nos que os denominadores nas divisões têm termo constante 1 (bastaria ser diferente de zero). Como estamos supondo que a característica do corpo K é zero, para cada variável x_i $\exists g_i \in K$ tal que, substituindo x_i por $x_i - g_i$, os denominadores são unidades na série formal de potências $K[[x_1, \dots, x_n]]$.

Teorema 2.2 *Seja F um conjunto de formas polinomiais em $K[x_1, \dots, x_n]$, cada uma com grau $\leq g$. Existe um algoritmo $\mathcal{A}$ que computa F , sem usar divisões, e:*

$$D(F) \leq \frac{g(g-1)}{2} D(\mathcal{A})$$

Demonstração .- Seja $A' = (p'_1, \dots, p'_m)$ um algoritmo ótimo para F . Novamente consideramos $L_i : i = 0, \dots, g$, como sendo o operador linear sobre o

espaço dos polinômios que dá o termo homogêneo de grau i , e L o operador que dado um polinômio nos dá os termos de grau $\leq g$ do polinômio, i.e., $L = L_0 + \dots + L_g$.

Construamos o algoritmo $\mathcal{A}$ mantendo os passos de A' que não são divisões e substituindo as divisões conforme apresentaremos a seguir.

Seja $p'_j = h/i$ onde $h, i < j$. Como $p_i = (p_i/p_j) * p_j$, temos que para qualquer grau $d \leq g$:

$$L_d(p_i) = \sum_{l+m=d} L_l(p_i/p_j) * L_m(p_j) = \sum_{l=0}^{d-1} L_l(p_i/p_j) * L_{d-l}(p_j) + L_d(p_i/p_j) * L_0(p_j)$$

Como estamos supondo que os denominadores p_j têm termo constante 1, $L_0(p_j)$ vale 1, e então:

$$\boxed{L_d(p_i/p_j) = L_d(p_i) - \sum_{l=0}^{d-1} L_l(p_i/p_j) * L_{d-l}(p_j)}$$

Operando:

$$\begin{aligned} L_0(p'_h) &= L_0(p_i) \\ L_1(p'_h) &= L_1(p_i) - L_0(p'_h) * L_1(p_j) \\ L_2(p'_h) &= L_2(p_i) - L_0(p'_h) * L_2(p_j) - L_1(p'_h) * L_1(p_j) \\ &\vdots \\ L_g(p'_h) &= L_g(p_i) - L_0(p'_h) * L_g(p_j) - \dots - L_{g-1}(p'_h) * L_1(p_j) \end{aligned}$$

Somando, temos que:

$$\begin{aligned} p_h &= L(p'_h) = L_0(p'_h) + \dots + L_g(p'_h) \\ &= \sum_{l=0}^g L_l(p_i) - L_0(p'_h) * \sum_{l=1}^g L_l(p_j) - L_1(p'_h) * \sum_{l=1}^{g-1} L_l(p_j) - \dots \\ &\quad \dots - L_{g-2}(p'_h) * \sum_{l=1}^2 L_l(p_j) - L_{g-1}(p'_h) * L_1(p_j) \end{aligned}$$

Para determinar o número de produtos em p_h , suponhamos que $L_i(p'_h)$ para $i = 0, \dots, g$, estão armazenados em variáveis auxiliares de memória. Vejamos

quantos produtos são necessários.

Para o cálculo de $L_0(p'_h)$ e $L_1(p'_h)$ não foram usadas multiplicações essenciais. Para $L_2(p'_h)$ temos um produto; para $L_3(p'_h)$, dois produtos e assim sucessivamente, para $L_{g-1}(p'_h)$ temos $g - 2$ produtos. Para calcular p_h temos mais $g - 1$ multiplicações, portanto podemos substituir uma divisão por $\frac{g(g-1)}{2}$ produtos. $\square$

Corolário 2.1 *Se $F \subseteq K[x_1, \dots, x_n]$ é um conjunto de formas quadráticas então existe um algoritmo ótimo para F que não usa divisões.*

Observações :

- 1) Se tivéssemos um método direto para obter $L_i(p'_h)$ $i = 0, \dots, g$, teríamos que cada divisão poderia ser substituída por $g - 1$ produtos.
- 2) Strassen [55] e De Groote [20] dão outra forma de substituir uma divisão por $\frac{g(g-1)}{2}$ produtos. Sieveking [46] usando algoritmos rápidos para a divisão de séries de potências formais, reduz a ordem para:

$$D(F) \leq 5.75 g D(A)$$

2.3.4 Cotas Inferiores

Todo algoritmo que resolve um problema dado, nos dá uma cota superior da complexidade desse problema. Procuramos algoritmos que melhorem a complexidade conhecida de um problema. O limite para essas melhoras está dado por uma cota inferior do problema, i.e., o mínimo necessário para resolver um problema, independentemente do algoritmo usado. Como essa cota independe do algoritmo, devemos considerar as características intrínsecas do problema e aproveitar a sua estrutura matemática.

O teorema abaixo é fundamental. Ele prova que a complexidade de qualquer subconjunto de um anel de polinômios, está limitada inferiormente pela sua dimensão.

Teorema 2.3 $\forall T \subseteq K[x_1, \dots, x_n] : D(T) \geq \dim \langle T \rangle$.

Demonstração .- Seja $\mathcal{A} = (p_1, \dots, p_m)$ um algoritmo que computa o conjunto $T = \{q_1, \dots, q_t\} \subseteq K[x_1, \dots, x_n]$, e sejam $(h_1, \dots, h_l)$ suas multiplicações essenciais.

i) Pela definição 2.6 : $T \subseteq \langle \mathcal{A} \rangle$

ii) Pela proposição 2.5 : $q_i \in \langle h_1, \dots, h_l \rangle \quad \forall i = 1, 2, \dots, t$.

Então :

$$\langle T \rangle \subseteq \langle h_1, \dots, h_l \rangle \Rightarrow \dim \langle T \rangle \leq \dim \langle h_1, \dots, h_l \rangle$$

Como $\mathcal{A}$ é arbitrário, temos que :

$$\boxed{\dim \langle T \rangle \leq D(T)}$$

Corolário 2.2 *O produto no espaço $\mathcal{P}_n$ dos polinômios numa variável, tem complexidade $2n + 1$.*

Demonstração .- Seja C o conjunto dos coeficientes do produto de dois polinômios quaisquer de grau n . Como C é um conjunto linearmente independente:

$$D(\mathcal{P}_n) \geq \dim \langle C \rangle = 2n + 1$$

Como o algoritmo de Winograd para o produto de dois polinômios numa variável, da seção 2.2.1, tem complexidade $2n + 1$, temos que:

$$\boxed{D(\mathcal{P}_n) = 2n + 1}$$

O problema de multiplicar n polinômios, $n \geq 2$, é estudado no capítulo 4. Interessa-nos ter uma cota inferior deste problema. Sejam: m_i o grau do i -ésimo polinômio a ser multiplicado, $m = \sum_{i=1}^n m_i$ o grau do polinômio resultante, e $z_0, \dots, z_m$ os coeficientes do produto. Como $\{z_0, \dots, z_m\}$ é um conjunto linearmente independente, temos:

Corolário 2.3 *O produto de n polinômios numa variável de graus m_i , $i = 1, \dots, n$, tem cota inferior:*

$$\sum_{i=1}^n m_i + 1$$

Demonstração .- Análoga à anterior. $\square$

Também queremos ter uma cota inferior para o produto de 2 polinômios, p e q , em m variáveis. Sejam $x_1, \dots, x_m$ as indeterminadas, e n_i a potência máxima em x_i , $\forall i$, nos dois polinômios. O polinômio produto de $p(x_1, \dots, x_m)$ e $q(x_1, \dots, x_m)$ tem $2n_i$ como potência máxima em x_i , $\forall i$, e:

$$\prod_{i=1}^m (2n_i + 1)$$

coeficientes. Analogamente aos corolários anteriores, como o conjunto de coeficientes é linearmente independente, temos:

Corolário 2.4 *O produto de 2 polinômios em m variáveis $x_1, \dots, x_m$, com n_i como potência máxima na indeterminada x_i , $\forall i$, nos dois polinômios, tem cota inferior:*

$$\prod_{i=1}^m (2n_i + 1)$$

Capítulo 3

Teoria Multilinear

No capítulo 1, mostramos que a operação de produto numa álgebra qualquer tem como resultado um conjunto de formas bilineares. Considerando cada elemento da álgebra como uma forma linear, os coeficientes do produto de m elementos da álgebra, dão como resultado um conjunto de formas m – lineares. Exemplos desta classe de problemas, são: produtos da mesma matriz m vezes (usado em caminhos de comprimento menor ou igual a m de um grafo, polinômio característico, etc.), produto de polinômios numa ou mais variáveis, etc.

No capítulo anterior vimos exemplos de algoritmos algébricos que computam polinômios quaisquer. Consideremos agora polinômios multilineares, isto é, polinômios homogêneos nos quais todos os monômios têm o mesmo ‘nome’. Vamos dar uma definição de algoritmo mais específica, para este tipo de polinômios. Antes disso, tornaremos preciso o conceito de nome e vamos ver alguns exemplos de polinômios multilineares.

Sejam X o conjunto de indeterminadas, e N um conjunto cujos elementos serão os nomes, e suponha dada uma função (em geral não injetora):

$$\varphi : X \rightarrow N$$

que associa a cada indeterminada o seu nome. Se chamamos de $M(X)$ e $M(N)$ aos monóides dos monômios sobre X e N , respectivamente, φ se es-

tende naturalmente a um morfismo:

$$\varphi : M(\mathbf{X}) \rightarrow M(N)$$

Um monômio $w \in M(\mathbf{X})$ é multilinear se todas as indeterminadas em w têm nomes distintos. Um polinômio é *multilinear* se φ tem o mesmo valor nos monômios de seu suporte e todos os monômios são multilineares.

Exemplos :

1) seja $X = \{x_1, x_2, \dots, y_1, y_2, \dots, w_1, w_2, \dots\}$ e $N = \{x, y, \dots, w\}$, considere:

$$\alpha_i \xrightarrow{\varphi} \alpha$$

Os polinômios seguintes são multilineares :

- i) $x_1 + x_2$
- ii) $x_1 y_1 - x_2 y_2$
- iii) $x_1 y_1 z_1$

enquanto que os polinômios abaixo não são multilineares.

- i) $1 + x_1$
- ii) x_1^2
- iii) $x_1 + y_2$
- iv) $x_1 y_1 z_1 + x_1 y_1$

2) seja $X = \{x_{ij} : 1 \leq i \leq n, 1 \leq j \leq m_i\}$ e $N = \{i : x_{ij} \text{ é indeterminada}\}$, considere:

$$x_{ij} \xrightarrow{\varphi} i$$

Alguns polinômios multilineares são :

- i) $x_{11} + x_{12} + x_{13}$
- ii) $x_{11}x_{21} + x_{12}x_{22} + x_{13}x_{23}$
- iii) $x_{11}x_{21}x_{32}x_{42} + x_{12}x_{22}x_{31}x_{41}$

Não são multilineares :

- i) $1 + x_{11}$
- ii) $x_{11}x_{12}$
- iii) $x_{11} + x_{21}$

Sempre trabalharemos com uma *matriz genérica de indeterminadas* $X_{n \times r}$. Isto quer dizer que as variáveis são: x_{ij} onde $i = 1, \dots, n$ e $j = 1, \dots, r$, e que x_i é o nome da variável. É fácil de comprovar que outros casos (quantidades de variáveis distintas para nomes distintos), podem ser expressos em função da matriz genérica de indeterminadas.

3.1 Algoritmo e Complexidade Multilinear

Vamos agora definir algoritmos multilineares. Para um algoritmo algébrico ser multilinear, exigiremos que todos os polinômios associados aos seus passos sejam multilineares (sobre álgebra multilinear pode-se consultar Greub [27]). Visando este objetivo, não devemos permitir:

- 1) uso de constantes fora de produtos por escalares
- 2) soma de variáveis de nomes distintos
- 3) soma de passos anteriores com nomes diferentes
- 4) produto de passos anteriores com algum nome em comum

Num algoritmo multilinear, da mesma forma que para um algoritmo algébrico geral, os passos serão ou básicos ou somas ou multiplicações.

Os básicos são as indeterminadas. As somas são combinações lineares dos passos anteriores, tais que todos seus monômios têm igual nome e cada um deles verifica não ter produtos de mesmo nome. Caso contrário, os monômios dessa combinação linear não seriam homogêneos, e daí o resultado não seria multilinear.

As multiplicações são produtos de dois passos anteriores, respeitando ter variáveis com nomes diferentes. O nome do resultado do produto é a união dos nomes dos fatores.

Definição 3.1 Um algoritmo algébrico é *multilinear* se os polinômios associados aos seus passos são multilineares.

É fácil ver que a definição de algoritmo multilinear corresponde à seguinte caracterização:

i) se $p_j \in K^{j-1}$ tal que $p_j = (a_1, \dots, a_{(j-1)})$ então $\forall k, l$ tal que $a_k \neq 0$ e $a_l \neq 0$:

$$\varphi(p'_k) = \varphi(p'_l)$$

ii) se $p_j = (i, h)$ onde $i, h < j$ então :

$$\begin{aligned} \varphi(p'_i) &\neq \emptyset \\ \varphi(p'_h) &\neq \emptyset \\ \varphi(p'_i) \cap \varphi(p'_h) &= \emptyset \end{aligned}$$

Para um algoritmo multilinear $\mathcal{A}$ definimos a *complexidade multilinear* como sendo o número de multiplicações de $\mathcal{A}$. Analogamente à definição 2.6, temos:

Definição 3.2 Seja $T \subseteq K[X]$ um conjunto finito de formas multilineares. A complexidade *multilinear* de T é definida por:

$$M(T) := \min \{D(A) : T \subseteq \langle A \rangle \text{ e } A \text{ é um algoritmo multilinear}\}$$

Interessa vincular a complexidade algébrica à complexidade multilinear. Obviamente, como todo algoritmo multilinear é algébrico, $D(T) \leq M(T)$. Uma questão em aberto é saber se, para todo conjunto finito de formas multilineares T , $D(T) = M(T)$, mas a seguinte proposição dá uma relação entre as duas complexidades.

Proposição 3.1 Para cada conjunto finito $T \subseteq K[x_{11}, \dots, x_{nr}]$ de formas multilineares:

$$D(T) \leq M(T) \leq 3^m D(T)$$

Demonstração .- A partir do conjunto $N = \{1, 2, \dots, n\}$ consideremos os conjuntos N_0 dos nomes multilineares, i.e., $N_0 = \mathcal{P}(N)$, e N_1 dos nomes não multilineares. Se $w \in N_0$ e p é um polinômio em $K[x_{11}, \dots, x_{nr}]$, chamaremos p_w ao polinômio multilinear formado pelos monômios em p de nome w . Seja $\mathcal{A}$ um algoritmo algébrico ótimo para T com passos $(p_1, \dots, p_l)$. Note que cada passo p de $\mathcal{A}$ pode ser escrito como:

$$p = \sum_{w \in N_0} p_w + \sum_{y \in N_1} p_y$$

A partir de $\mathcal{A}$ construiremos um algoritmo multilinear com a seguinte propriedade: cada passo p do algoritmo $\mathcal{A}$ é substituído pela seqüência formada por todos os passos p_w , com $w \in N_0$. Portanto p transforma-se em $p_1, \dots, p_s$, onde s só depende do número de nomes possíveis. A seguir damos a construção anterior. Os passos básicos do algoritmo $\mathcal{A}$ ficam iguais no algoritmo multilinear. As somas são alteradas substituindo cada somando p pelos seus correspondentes p_w , com $w \in N_0$, já transformados em passos anteriores. Agora, seja $p = qr$ um produto no algoritmo algébrico $\mathcal{A}$. Temos que calcular todos os p_w , $w \in N_0$. Para isso:

$$p_w = (qr)_w = \sum_{uv=w} q_u r_v$$

Se w contém k elementos, calcular p_w tem um custo de 2^k , correspondente ao número de fatorações de w em uv . Logo, para calcular *todos* os p_w temos um custo de:

$$\sum_k \binom{m}{k} 2^k = 3^m$$

Portanto:

$$M(T) \leq 3^m D(T)$$

3.2 Exemplos

Nesta subseção, vamos ver que os exemplos 2 e 3 considerados na seção 2.2, são multilineares e vamos dar um exemplo de um algoritmo não multilinear para o produto de três números complexos. O nome do polinômio associado ao passo p_i será denotado por n_i .

1) Produto de três números complexos trilinear

O produto dos três números complexos : $x_1 + ix_2, y_1 + iy_2, z_1 + iz_2$ é :

$$(x_1y_1z_1 - x_1y_2z_2 - x_2y_1z_2 - x_2y_2z_1) + i(x_1y_1z_2 + x_1y_2z_1 + x_2y_1z_1 - x_2y_2z_2)$$

O algoritmo multilinear correspondente ao algoritmo da subseção 2.2 é dado abaixo segundo as duas definições exibidas:

x_1	p_1	=	x_1
	n_1	=	$\{1\}$
x_2	p_2	=	x_2
	n_2	=	$\{1\}$
y_1	p_3	=	y_1
	n_3	=	$\{2\}$
y_2	p_4	=	y_2
	n_4	=	$\{2\}$
z_1	p_5	=	z_1
	n_5	=	$\{3\}$
z_2	p_6	=	z_2
	n_6	=	$\{3\}$

M_1	$:=$	$x_1 * y_1$	p_7	$=$	$(1, 3)$
M_2	$:=$	$x_2 * y_2$	n_7	$=$	$n_1 \cup n_3 = \{1, 2\}$
S_3	$:=$	$x_1 + x_2$	p_8	$=$	$(2, 4)$
S_4	$:=$	$y_1 + y_2$	n_8	$=$	$n_2 \cup n_4 = \{1, 2\}$
M_5	$:=$	$S_3 * S_4$	p_9	$=$	$(1, 1, 0, 0, 0, 0, 0, 0)$
ϕ_6	$:=$	$M_1 - M_2$	n_9	$=$	$\{1\}$
ϕ_7	$:=$	$M_5 - M_1 - M_2$	p_{10}	$=$	$(0, 0, 1, 1, 0, 0, 0, 0)$
M_8	$:=$	$\phi_6 * z_1$	n_{10}	$=$	$\{2\}$
M_9	$:=$	$\phi_7 * z_2$	p_{11}	$=$	$(9, 10)$
S_{10}	$:=$	$\phi_6 + \phi_7$	n_{11}	$=$	$n_9 \cup n_{10} = \{1, 2\}$
S_{11}	$:=$	$z_1 + z_2$	p_{12}	$=$	$(0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0)$
M_{12}	$:=$	$S_{10} * S_{11}$	n_{12}	$=$	$n_7 = n_8 = \{1, 2\}$
f_1	$:=$	$M_8 - M_9$	p_{13}	$=$	$(0, 0, 0, 0, 0, 0, -1, -1, 0, 0, 1, 0)$
f_2	$:=$	$M_{12} - M_8 - M_9$	n_{13}	$=$	$n_{11} = n_7 = n_8 = \{1, 2\}$
			p_{14}	$=$	$(12, 5)$
			n_{14}	$=$	$n_{12} \cup n_5 = \{1, 2, 3\}$
			p_{15}	$=$	$(13, 6)$
			n_{15}	$=$	$n_{13} \cup n_6 = \{1, 2, 3\}$
			p_{16}	$=$	$(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0)$
			n_{16}	$=$	$n_{12} = n_{13} = \{1, 2\}$
			p_{17}	$=$	$(0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0)$
			n_{17}	$=$	$\{3\}$
			p_{18}	$=$	$(16, 17)$
			n_{18}	$=$	$n_{16} \cup n_{17} = \{1, 2, 3\}$
			p_{19}	$=$	$(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0)$
			n_{19}	$=$	$n_{14} = n_{15} = \{1, 2, 3\}$
			p_{20}	$=$	$(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1, 0, 0, 1, 0)$
			n_{20}	$=$	$n_{18} = n_{14} = n_{15} = \{1, 2, 3\}$

O algoritmo respeita as condições de multilinearidade . Os seus produtos estão nos passos : 5, 6, 9, 14, 15 e 18, e portanto a sua complexidade é 6. $\square$

2) Cômputo da função :

$$f := x_1 y_1 z_1 w_1 + x_1 y_1 z_2 w_2 + x_2 y_2 z_1 w_1 + x_2 y_2 z_2 w_2$$

Abaixo damos o algoritmo multilinear correspondente à fatoração da fórmula anterior em:

$$(x_1 y_1 + x_2 y_2) * (z_1 w_1 + z_2 w_2)$$

x_1		p_1	$=$	x_1	
		n_1	$=$	$\{1\}$	
x_2		p_2	$=$	x_2	
		n_2	$=$	$\{1\}$	
y_1		p_3	$=$	y_1	
		n_3	$=$	$\{2\}$	
y_2		p_4	$=$	y_2	
		n_4	$=$	$\{2\}$	
z_1		p_5	$=$	z_1	
		n_5	$=$	$\{3\}$	
z_2		p_6	$=$	z_2	
		n_6	$=$	$\{3\}$	
w_1		p_7	$=$	w_1	
		n_7	$=$	$\{4\}$	
w_2		p_8	$=$	w_2	
		n_8	$=$	$\{4\}$	
M_1	$:=$	$x_1 * y_1$	p_9	$=$	$(1, 3)$
			n_9	$=$	$n_1 \cup n_3 = \{1, 2\}$
M_2	$:=$	$x_2 * y_2$	p_{10}	$=$	$(2, 4)$
			n_{10}	$=$	$n_2 \cup n_4 = \{1, 2\}$
M_3	$:=$	$z_1 * w_1$	p_{11}	$=$	$(5, 7)$
			n_{11}	$=$	$n_5 \cup n_7 = \{3, 4\}$
M_4	$:=$	$z_2 * w_2$	p_{12}	$=$	$(6, 8)$
			n_{12}	$=$	$n_6 \cup n_8 = \{3, 4\}$
S_5	$:=$	$M_1 + M_2$	p_{13}	$=$	$(0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0)$
			n_{13}	$=$	$n_9 = n_{10} = \{1, 2\}$
S_6	$:=$	$M_3 + M_4$	p_{14}	$=$	$(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0)$
			n_{14}	$=$	$n_{11} = n_{12} = \{3, 4\}$
f	$:=$	$S_5 * S_6$	p_{15}	$=$	$(13, 14)$
			n_{15}	$=$	$n_{13} \cup n_{14} = \{1, 2, 3, 4\}$

O algoritmo é multilinear e sua complexidade é 5. Os seus passos multiplicativos são : 9,10,11,12 e 15. $\square$

3) Produto de três complexos não trilinear

x_1	p_1	$=$	x_1
	n_1	$=$	$\{1\}$
x_2	p_2	$=$	x_2
	n_2	$=$	$\{1\}$
y_1	p_3	$=$	y_1
	n_3	$=$	$\{2\}$
y_2	p_4	$=$	y_2
	n_4	$=$	$\{2\}$

z_1		$p_5 = z_1$
z_2		$n_5 = \{3\}$
$M_1 := x_1 * x_2$		$p_6 = z_2$
$M_2 := y_1 * y_2$		$n_6 = \{3\}$
$M_3 := x_1 * y_1$		$p_7 = (1, 2)$
$M_4 := x_2 * y_2$		$n_7 = \text{indefinido: } n_1 \cap n_2 \neq \emptyset$
$S_1 := x_1 + y_1$		$p_8 = (3, 4)$
$S_2 := x_2 + y_2$		$n_8 = \text{indefinido: } n_3 \cap n_4 \neq \emptyset$
$M_5 := S_1 * S_2$		$p_9 = (1, 3)$
$S_3 := M_3 - M_4$		$n_9 = n_1 \cup n_3 = \{1, 2\}$
$S_4 := M_5 - M_1 - M_2$		$p_{10} = (2, 4)$
$M_6 := S_3 * S_4$		$n_{10} = n_2 \cup n_4 = \{1, 2\}$
$M_7 := z_1 * z_2$		$p_{11} = (1, 0, 1, 0, 0, 0)$
$M_8 := S_3 * z_1$		$n_{11} = \text{indefinido: variáveis com nomes diferentes}$
$M_9 := S_4 * z_2$		$p_{12} = (0, 1, 0, 1, 0, 0)$
$S_5 := S_3 + z_1$		$n_{12} = \text{indefinido: variáveis com nomes diferentes}$
$S_6 := S_4 + z_2$		$p_{13} = (11, 12)$
$M_{10} := S_5 * S_6$		$n_{13} = \text{indefinido: variáveis com nomes diferentes}$
$f_1 := M_8 - M_9$		$p_{14} = (0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0)$
$f_2 := M_{10} - M_6 - M_7$		$n_{14} = n_9 = n_{10} = \{1, 2\}$
		$p_{15} = (0, 0, 0, 0, 0, 0, -1, -1, 0, 0, 0, 0, 1, 0)$
		$n_{15} = \text{indefinido: variáveis com nomes diferentes}$
		$p_{16} = (14, 15)$
		$n_{16} = \text{indefinido: } n_{14} \cap n_{15} \neq \emptyset$
		$p_{17} = (5, 6)$
		$n_{17} = \text{indefinido: } n_5 \cap n_6 \neq \emptyset$
		$p_{18} = (14, 5)$
		$n_{18} = n_{14} \cup n_5 = \{1, 2, 3\}$
		$p_{19} = (15, 6)$
		$n_{19} = \text{indefinido: variáveis com nomes diferentes}$
		$p_{20} = (0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$
		$n_{20} = \text{indefinido: variáveis com nomes diferentes}$
		$p_{21} = (0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$
		$n_{21} = \text{indefinido: variáveis com nomes diferentes}$
		$p_{22} = (20, 21)$
		$n_{22} = \text{indefinido: variáveis com nomes diferentes}$
		$p_{23} = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0)$
		$n_{23} = \text{indefinido: variáveis com nomes diferentes}$
		$p_{24} = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1, 0, 0, 0, 0, 1, 0)$
		$n_{24} = \text{indefinido: variáveis com nomes diferentes}$

O algoritmo efetivamente computa o produto de três complexos, pois:

$$\begin{aligned}
f_1 &= M_8 - M_9 \\
&= S_3 * z_1 - S_4 * z_2 \\
&= (M_3 - M_4) * z_1 - (M_5 - M_1 - M_2) * z_2 \\
&= x_1 y_1 z_1 - x_1 y_2 z_2 - x_2 y_1 z_2 - x_2 y_2 z_1
\end{aligned}$$

$$\begin{aligned}
f_2 &= M_{10} - M_6 - M_7 \\
&= S_5 * S_6 - S_3 * S_4 - z_1 * z_2 \\
&= (S_3 + z_1) * (S_4 + z_2) - S_3 * S_4 - z_1 * z_2 \\
&= S_3 * z_2 + S_4 * z_1 \\
&= (M_3 - M_4) * z_2 - (M_5 - M_1 - M_2) * z_1 \\
&= x_1 y_1 z_2 + x_1 y_2 z_1 + x_2 y_1 z_1 - x_2 y_2 z_2
\end{aligned}$$

Porém o algoritmo não é multilinear. Nos passos : 5, 6, 9, 10, 11, 13, 16, 17, 19, 20, 21, 22, 23 e 24, o nome correspondente não está definido. Um nome indefinido carrega a sua indefinição para todos os passos seguintes que usem esse passo.

As regras violadas, e os passos onde aconteceram as violações, foram:

- 1) interseção de nomes não vazia : 5,6,17;
- 2) soma de variáveis com distintos nomes : 9,10,20;
- 3) uso de nomes indefinidos : 11,13,16,19,21,22,23,24.

3.3 Gerador de Algoritmos Multilineares

Conhecemos algoritmos ótimos para o produto de dois polinômios, de dois números complexos, etc. Se iteramos esses algoritmos obtemos algoritmos multilineares para o produto de n polinômios ou n números complexos. Queremos estudar qual das formas possíveis de iteração é a melhor.

Observe que não vamos ter necessariamente como resultado um algoritmo ótimo, mas teremos certeza de que dentre os algoritmos que surgem de iterar o algoritmo binário, ficamos com o que tem complexidade mínima. Nesse sentido ele será quase-ótimo.

Para determinar qual a melhor forma para iterar, vamos aplicar uma programação dinâmica:

$$D(n) = \min_k \{ D(k) + D(n - k) + \text{custo} * \}$$

onde $D(i)$ é a complexidade i -linear, e $\text{custo} *$ é o custo de operar, em forma ótima, as formas k e $n - k$ lineares.

Como a operação $*$ é binária, a decomposição em duas partes é correta; estamos tomando a melhor forma de operar com as partes de tamanho k e $n - k$ (dada por $D(k)$ e $D(n - k)$), que assegura-nos complexidade mínima para os produtos de k e $n - k$ elementos respectivamente.

Exemplo : Produto de números complexos

Dados dos números complexos: $x_1 + x_2i$ e $y_1 + y_2i$, seu produto é:

$$(x_1 + x_2i)(y_1 + y_2i) = (x_1y_1 - x_2y_2) + (x_1y_2 + x_2y_1)i$$

Podemos formular o problema de multiplicar dois números complexos da seguinte forma : dados $(x_1, x_2), (y_1, y_2) \in \mathbb{Q}^2$, computar as duas formas bilineares $(x_1y_1 - x_2y_2, x_1y_2 + x_2y_1) \in \mathbb{Q}^2$.

O método dado pela definição do produto de dois números complexos utiliza 4 produtos. Winograd [63] desenvolveu um algoritmo com complexidade 3 para o produto de dois números complexos, provando também, que o algoritmo é ótimo. Damos abaixo o algoritmo de Winograd.

x_1		p_1	$=$	x_1
x_2		p_2	$=$	x_2
y_1		p_3	$=$	y_1
y_2		p_4	$=$	y_2
z_1		p_5	$=$	z_1
z_2		p_6	$=$	z_2
M_1	$:=$	$x_1 * y_1$		$(1, 3)$
M_2	$:=$	$x_2 * y_2$		$(2, 4)$
S_3	$:=$	$x_1 + x_2$		$(1, 1, 0, 0, 0, 0, 0, 0)$
S_4	$:=$	$y_1 + y_2$		$(0, 0, 1, 1, 0, 0, 0, 0)$
M_5	$:=$	$S_3 * S_4$		$(9, 10)$
f_1	$:=$	$M_1 - M_2$		$(0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0)$
f_2	$:=$	$M_5 - M_1 - M_2$		$(0, 0, 0, 0, 0, 0, -1, -1, 0, 0, 1, 0)$

Consideremos o problema de multiplicar n números complexos.
A complexidade de um algoritmo que utiliza uma estratégia tail recursion está definida pela recorrência:

$$\begin{aligned} D(2) &= 3 \\ D(n) &= D(n-1) + 3, \quad n > 2 \end{aligned}$$

Se usarmos um algoritmo para o produto de n números complexos que aplique divisão e conquista, teremos que sua complexidade é dada por:

$$\begin{aligned} D(2) &= 3 \\ D(n) &= D\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + D\left(\left\lceil \frac{n}{2} \right\rceil\right) + 3, \quad n > 2 \end{aligned}$$

Para escolher qual é a melhor forma de iterar o algoritmo ótimo para o produto de 2 números complexos, temos que resolver a recorrência:

$$\begin{aligned} D(2) &= 3 \\ D(n) &= \min_k \{ D(k) + D(n-k) + 3 \} \end{aligned}$$

Proposição 3.2 *A solução das três recorrências anteriores é: $3(n - 1)$.*

Demonstração .- Por indução em n . $\square$

Implícito no resultado anterior está uma consequência prática importante: todas as estratégias usadas para iterar o produto de dois números complexos têm a mesma complexidade !

Na realidade, provamos algo mais importante: toda vez que se itere um algoritmo cuja complexidade é constante, teremos como resultado que todas as estratégias têm igual complexidade. Isto é, provamos a seguinte proposição:

Proposição 3.3 *Se a complexidade ótima do produto numa álgebra qualquer $\mathcal{A}$ é constante, c , então todas as formas de iterar o algoritmo ótimo para o produto de n elementos de $\mathcal{A}$ têm a mesma complexidade $c(n - 1)$.*

Algumas álgebras com produtos constantes são: extensões simples de R e Q ($Q(\sqrt{2})$, C , etc), quatérnios, etc.

Capítulo 4

Produto de Polinômios

Neste capítulo analisaremos o problema de multiplicar n polinômios, com $n \geq 2$. Dividimos o estudo em duas seções: polinômios numa variável e polinômios em várias variáveis. Damos algoritmos para cada um dos casos e mostramos suas complexidades, relacionando-as com as complexidades ótimas. O resultado mais importante, dá uma classe de polinômios para os quais obtemos algoritmos assintoticamente ótimos.

4.1 Produto de Polinômios Numa Variável

Sejam:

$$f_i(x) = \sum_{j=0}^{m_i} a_{ij}x^j \quad 1 \leq i \leq n$$

n polinômios numa variável x . Consideremos a_{ij} $1 \leq i \leq n$, $0 \leq j \leq m_i$ como indeterminadas. Calculando diretamente o coeficiente em x^l :

$$\sum_{k_1 + \dots + k_n = l} \left(\prod_{i=1}^n a_{ik_i} \right)$$

temos que o número de produtos é:

$$\binom{l}{n} \cdot (n-1) = \frac{l!}{n! (l-n)!} \cdot (n-1) = \frac{l(l-1) \cdots (l-n+1)}{(n-2)! \cdot n} = O(l^n)$$

Para calcular o coeficiente de x^l , temos um custo da ordem de l^n , onde: $0 \leq l \leq \sum_{i=1}^n m_i$. Portanto cálculos diretos são inviáveis. Melhoraremos esta ordem aplicando a programação dinâmica descrita na seção 3.3.

4.1.1 Produto de polinômios com mesmo grau

Damos a seguir alguns algoritmos para o produto de vários polinômios de grau comum m , junto com suas respectivas complexidades. Trabalharemos em uma linguagem tipo pseudo-Pascal.

Algoritmo 1 : Tail Recursion

Chamaremos 'Prod' ao procedimento que computa o produto de dois polinômios de forma ótima, segundo o algoritmo de Winograd da seção 2.2. A função que multiplica os polinômios tem como parâmetros um conjunto cujos elementos são os polinômios que compõem o produto e o número de polinômios a ser multiplicados. O resultado da função é o polinômio com $nm + 1$ coeficientes correspondente ao produto dos polinômios da entrada.

```
função Mult (A : polinômios; n : inteiro) : polinômio;
se n  $\neq$  1 então
    se n=2 então Mult := Prod (A[1] , A[2])
    senão Mult := Prod (Mult(A,n-1) , A[n])
```


Proposição 4.1 *O custo do algoritmo 1 é $O(n^2m)$. Mais precisamente, temos :*

$$D(n) = (n - 1) \left[\frac{m(n + 2)}{2} + 1 \right]$$

Demonstração .- A complexidade do algoritmo está definida pela recorrência :

$$\begin{aligned} D(1) &= 0 \\ D(n) &= D(n-1) + (nm + 1), \quad n > 2 \end{aligned}$$

A prova será feita por indução em n .

Base :

$$D(1) = (1 - 1) \left[\frac{m(1 + 2)}{2} + 1 \right] = 0$$

Passo :

$$\begin{aligned} D(n) &= D(n - 1) + (nm + 1) \\ &= (n - 2) \left[\frac{m(n + 1)}{2} + 1 \right] + (nm + 1) \\ &= m \frac{(n + 1)(n - 2)}{2} + nm + (n - 1) \\ &= m \left(\frac{n^2 - n - 2}{2} + n \right) + (n - 1) \\ &= m \left(\frac{n^2 + n - 2}{2} \right) + (n - 1) \\ &= m \frac{(n - 1)(n + 2)}{2} + (n - 1) \\ &= (n - 1) \left[\frac{m(n + 2)}{2} + 1 \right] \end{aligned}$$

□

Algoritmo 2 : Divisão e Conquista

Podemos supor que o conjunto dos polinômios a serem multiplicados estão armazenados num vetor. Como o algoritmo é recursivo, os parâmetros da função são : o vetor que representa o conjunto dos polinômios e dois índices que referenciam o começo e o fim, no vetor dos polinômios, dos polinômios multiplicados nessa iteração.

```
função Mult (A: polinômios; com, fim: inteiro): polinômio;  
se fim ≠ com  
então se fim-com = 1  
    então Mult := Prod (A[com] , A[fim])  
    senão Mult := Prod ( Mult(A, com,  $\lfloor \frac{fim+com}{2} \rfloor$ ) ,  
                        Mult(A,  $\lfloor \frac{fim+com}{2} \rfloor + 1$ , fim))
```

Para provar a ordem do algoritmo acima, necessitamos dos seguintes fatos:

Fato : *Para todo n*

$$\lceil \log n \rceil = \left\lceil \log \left\lfloor \frac{n}{2} \right\rfloor \right\rceil + 1$$
$$\lceil \log n \rceil = \begin{cases} \lceil \log(n+1) \rceil & \text{se } n \neq 2^l \\ \lceil \log(n+1) \rceil - 1 & \text{se } n = 2^l \end{cases}$$

Proposição 4.2 *O custo do algoritmo 2 é $O(mn \log n)$. Mais precisamente, temos :*

$$D(n) = (n-1)(m+1) + m \left(1 + n \lceil \log n \rceil - 2^{\lceil \log n \rceil} \right)$$

Demonstração .- No algoritmo 2, a complexidade está definida pela recorrência :

$$D(1) = 0$$

$$D(n) = D\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + D\left(\left\lceil \frac{n}{2} \right\rceil\right) + (nm + 1) \quad n > 1$$

A prova será feita por indução em n .

Base :

$$D(1) = (1 - 1)(m + 1) + m \left(1 + 1 \lceil \log 1 \rceil - 2^{\lceil \log 1 \rceil}\right) = 0$$

Passo : se $n = 2k$ temos que

$$\begin{aligned} D(n) &= 2D(k) + (2km + 1) \\ &= 2(k - 1)(m + 1) + m \left(2 + 2k \lceil \log k \rceil - 2^{\lceil \log k \rceil + 1}\right) + 2km + 1 \\ &= 4km + 2k - 1 + m \left(2k \lceil \log k \rceil - 2^{\lceil \log k \rceil + 1}\right) \\ &= 2nm + n - 1 + m \left[n (\lceil \log n \rceil - 1) - 2^{\lceil \log n \rceil - 1 + 1}\right] \\ &= (n - 1)(m + 1) + m \left(1 + n \lceil \log n \rceil - 2^{\lceil \log n \rceil}\right) \end{aligned}$$

Se $n = 2k + 1$ e $k \neq 2^l$, temos que:

$$\begin{aligned} D(n) &= D(k) + D(k + 1) + (2k + 1)m + 1 \\ &= (k - 1)(m + 1) + m \left(1 + k \lceil \log k \rceil - 2^{\lceil \log k \rceil}\right) + k(m + 1) \\ &\quad + m \left[1 + (k + 1) \lceil \log(k + 1) \rceil - 2^{\lceil \log(k + 1) \rceil}\right] + 2km + m + 1 \\ &= 4km + 2k + 2m + m \left[k \lceil \log k \rceil + (k + 1) \lceil \log(k + 1) \rceil\right. \\ &\quad \left.- 2^{\lceil \log k \rceil} - 2^{\lceil \log(k + 1) \rceil}\right] \\ &= 2nm + n - 1 + m \left[n (\lceil \log n \rceil - 1) - 2 \cdot 2^{\lceil \log n \rceil - 1}\right] \\ &= (n - 1)(m + 1) + m \left(1 + n \lceil \log n \rceil - 2^{\lceil \log n \rceil}\right) \end{aligned}$$

Se $n = 2k + 1$ e $k = 2^l$, temos que:

$$D(n) = D(k) + D(k + 1) + (2k + 1)m + 1$$

$$\begin{aligned}
&= (k-1)(m+1) + m \left(1 + k \lceil \log k \rceil - 2^{\lceil \log k \rceil} \right) + k(m+1) \\
&\quad + m \left[1 + (k+1) \lceil \log(k+1) \rceil - 2^{\lceil \log(k+1) \rceil} \right] + 2km + m + 1 \\
&= 4km + 2k + 2m + m \left[k \lceil \log k \rceil + (k+1) \lceil \log(k+1) \rceil \right. \\
&\quad \left. - 2^{\lceil \log k \rceil} - 2^{\lceil \log(k+1) \rceil} \right] \\
&= 2nm + n - 1 + m \left[2^l (\lceil \log n \rceil - 2) + (2^l + 1) (\lceil \log n \rceil - 1) \right. \\
&\quad \left. - 2^{\lceil \log n \rceil - 2} - 2^{\lceil \log n \rceil - 1} \right] \\
&= 2mn + n - 1 + m \left(n \lceil \log n \rceil - n - 2^l - \frac{3}{4} 2^{\lceil \log n \rceil} \right) \\
&= (n-1)(m+1) + m \left(1 + n \lceil \log n \rceil - 2^{\lceil \log n \rceil} + 2^{\lceil \log n \rceil - 2} - 2^l \right)
\end{aligned}$$

Como:

$$n = 2k + 1 = 2 \cdot 2^l + 1 = 2^{l+1} + 1$$

finalmente:

$$2^{\lceil \log n \rceil - 2} = 2^{(l+2)-2} = 2^l$$

Então:

$$D(n) = (n-1)(m+1) + m \left(1 + n \lceil \log n \rceil - 2^{\lceil \log n \rceil} \right)$$

Como:

$$n \leq 2^{\lceil \log n \rceil} < 2n$$

temos:

$$(n-1)(m+1) + m \left(1 + n \lceil \log n \rceil - 2^{\lceil \log n \rceil} \right) = O(mn \log n)$$

□

Consideremos, por exemplo, o produto de 4 polinômios de grau 1, isto é, $n = 4$ e $m = 1$. Sejam:

$$\begin{aligned}A(x) &= a_0 + a_1x \\B(x) &= b_0 + b_1x \\C(x) &= c_0 + c_1x \\D(x) &= d_0 + d_1x\end{aligned}$$

Seu produto é:

$$\begin{aligned}R(x) &= A(x)B(x)C(x)D(x) = (a_0 + a_1x)(b_0 + b_1x)(c_0 + c_1x)(d_0 + d_1x) \\&= a_0b_0c_0d_0 + (a_1b_0c_0d_0 + a_0b_1c_0d_0 + a_0b_0c_1d_0 + a_0b_0c_0d_1)x \\&\quad + (a_1b_1c_0d_0 + a_1b_0c_1d_0 + a_1b_0c_0d_1 + a_0b_1c_1d_0 + a_0b_1c_0d_1 + a_0b_0c_1d_1)x^2 \\&\quad + (a_1b_1c_1d_0 + a_1b_1c_0d_1 + a_1b_0c_1d_1 + a_0b_1c_1d_1)x^3 + a_1b_1c_1d_1x^4\end{aligned}$$

Primeiro analisemos alguns algoritmos que não usam o gerador de algoritmos quase-ótimos. O algoritmo que segue da definição do produto tem um custo de 48 (16 somandos, podendo calcular cada um com 3 produtos). Rapidamente melhoramos o algoritmo anterior considerando 8 variáveis auxiliares de memória para:

$$a_0b_0, a_0b_1, a_1b_0, a_1b_1, c_0d_0, c_0d_1, c_1d_0, c_1d_1$$

Assim para cada um dos 16 termos temos um produto, mais os 8 produtos armazenados na memória, a complexidade desse algoritmo é 24. Ainda melhor seria ter um algoritmo da forma divisão e conquista, mas multiplicando dois polinômios segundo o método da definição. Neste caso temos um custo de 17. Damos abaixo as operações efetuadas e os produtos calculados.

Polinômio	Custo	Produtos	Polinômio Resultado
$R^{(1)} = A.B$	4	$a_0b_0, a_0b_1, a_1b_0, a_1b_1$	$r_0^1 + r_1^1x + r_2^1x^2$
$R^{(2)} = C.D$	4	$c_0d_0, c_0d_1, c_1d_0, c_1d_1$	$r_0^2 + r_1^2x + r_2^2x^2$
$R = R^{(1)}.R^{(2)}$	9	$r_0^1r_0^2, \dots, r_2^1r_2^2$	$r_0 + r_1x + r_2x^2 + r_3x^3 + r_4x^4$
	17		

Consideremos agora os algoritmos 1 e 2 usando o algoritmo eficiente para o produto de dois polinômios. Lembramos que a complexidade do produto, utilizando este algoritmo, é igual à soma dos graus dos polinômios mais um.

Tail recursion .- Pela proposição 4.1, neste caso temos:

$$(n-1) \left[\frac{m(n+2)}{2} + 1 \right] = 3 \left(\frac{(1)(6)}{2} + 1 \right) = 12$$

A ordem em que são feitas as multiplicações e os custos associados são:

Produto	Custo
$R^{(1)} = A.B$	3 = $\partial A + \partial B + 1$
$R^{(2)} = R^{(1)}.C$	4 = $\partial R^{(1)} + \partial C + 1$
$R = R^{(2)}.D$	5 = $\partial R^{(2)} + \partial D + 1$
	<hr/> 12

Divisão e Conquista .- A complexidade do algoritmo, segundo a proposição 4.2, é dada por:

$$(n-1)(m+1)+m \left(1 + n \lceil \log n \rceil - 2^{\lceil \log n \rceil} \right) = (4-1)(1+1)+(1+4.2-2^2) = 11$$

Vejamos a ordem dos produtos neste caso.

Produto	Custo
$R^{(1)} = A.B$	3 = $\partial A + \partial B + 1$
$R^{(2)} = C.D$	3 = $\partial C + \partial D + 1$
$R = R^{(1)}.R^{(2)}$	5 = $\partial R^{(1)} + \partial R^{(2)} + 1$
	<hr/> 11

Utilizemos o gerador de algoritmo da seção 3.3. O polinômio produto de k polinômios de grau comum m tem grau km , e o polinômio produto de $n-k$ polinômios tem grau $(n-k)m$. Aplicando um algoritmo ótimo para o produto de dois polinômios de graus km e $(n-k)m$ (algoritmo 2.2.1), temos um

custo de : $km + (n - k)m + 1 = nm + 1$. Então a recorrência escreve-se:

$$D(1) = 0$$

$$D(n) = \min_k \{D(k) + D(n - k) + nm + 1\} \quad n > 1$$

Teorema 4.1 *A solução da recorrência acima é:*

$$D(n) = (n - 1)(m + 1) + m \left(1 + n \lceil \log n \rceil - 2^{\lceil \log n \rceil}\right)$$

Demonstração .- Chamemos:

$$T(n) = (n - 1)(m + 1) + m \left(1 + n \lceil \log n \rceil - 2^{\lceil \log n \rceil}\right)$$

Trivialmente: $T(1) = 0$.

Consideremos:

$$T(n) - T(n - 1) = (m + 1) + m \left(n \lceil \log n \rceil - (n - 1) \lceil \log(n - 1) \rceil - 2^{\lceil \log n \rceil} + 2^{\lceil \log(n - 1) \rceil} \right)$$

Se $\lceil \log n \rceil = \lceil \log(n - 1) \rceil$:

$$T(n) - T(n - 1) = (m + 1) + m \lceil \log n \rceil$$

Se $\lceil \log n \rceil \neq \lceil \log(n - 1) \rceil$ então $n = 2^l + 1$; neste caso:

$$\begin{aligned} T(n) - T(n - 1) &= (m + 1) + m \left[(2^l + 1)(l + 1) - 2^l \cdot l - 2^{l+1} + 2^l \right] \\ &= (m + 1) + m \left[2^l + l - 1 - 2^l \right] \\ &= (m + 1) + m \lceil \log n \rceil \end{aligned}$$

Portanto:

$$T(n) - T(n-1) = (m+1) + m \lceil \log n \rceil$$

Seja $f(k) = T(k) + T(n-k)$. Sem perda de generalidade, consideremos $k < \lfloor \frac{n}{2} \rfloor$. Mostraremos que f é decrescente, i.e., $f(k) \geq f(k+1)$.

$$\begin{aligned} f(k+1) - f(k) &= T(k+1) + T(n-k-1) - T(k) - T(n-k) \\ &= (T(k+1) - T(k)) - (T(n-k) - T(n-k-1)) \\ &= (m+1) + m \lceil \log(k+1) \rceil - (m+1) - m \lceil \log(n-k) \rceil \\ &= m (\lceil \log(k+1) \rceil - \lceil \log(n-k) \rceil) \end{aligned}$$

Como $k < \lfloor \frac{n}{2} \rfloor$, $k+1 < n-k$.

Portanto, $f(k+1) - f(k) \leq 0$ e:

$$\min_k \{ T(k) + T(n-k) \} = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right)$$

Então, resta provar que a solução da recorrência:

$$\begin{aligned} T(1) &= 0 \\ T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + (nm+1) \quad n > 1 \end{aligned}$$

é:

$$T(n) = (n-1)(m+1) + m \left(1 + n \lceil \log n \rceil - 2^{\lceil \log n \rceil}\right)$$

Como isso já foi demonstrado na proposição 4.2:

$$D(n) = (n-1)(m+1) + m \left(1 + n \lceil \log n \rceil - 2^{\lceil \log n \rceil}\right)$$

□

Corolário 4.1 *Para polinômios de mesmo grau, o melhor método é divisão e conquista.*

Demonstração .- Na prova do teorema anterior demonstramos que:

$$\min_k \{ D(k) + D(n-k) \} + nm + 1 = D\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + D\left(\left\lceil \frac{n}{2} \right\rceil\right) + (nm + 1)$$

□

Neste mesmo capítulo vamos ver que isto não é verdade quando os polinômios têm graus diferentes.

O corolário 2.3 dá uma cota inferior para o problema de multiplicar vários polinômios e o teorema 4.1 dá a solução que seria dada pelo gerador quase-ótimo, portanto, chamando P_m^n ao espaço formado pelo produto de n polinômios de grau comum m , temos:

$$nm + 1 \leq D(P_m^n) \leq (n-1)(m+1) + m \left(1 + n \lceil \log n \rceil - 2^{\lceil \log n \rceil}\right)$$

Uma pequena manipulação leva a uma fórmula mais expressiva:

$$nm \leq D(P_m^n) - 1 \leq nm \left(1 + \lceil \log n \rceil - \frac{2^{\lceil \log n \rceil}}{n} + \frac{1}{m} - \frac{2}{nm}\right)$$

Como $n \geq 2$, m inteiro positivo, e:

$$-2 \leq -\frac{2^{\lceil \log n \rceil}}{n} \leq -1$$

concluimos que:

$$nm \leq D(P_m^n) - 1 \leq nm(\lceil \log n \rceil - \varepsilon)$$

com $0 \leq \varepsilon < 1$.

4.1.2 Produto de polinômios com graus supercrescentes

Na seção anterior provamos que se todos os polinômios têm mesmo grau, divisão e conquista é o melhor método iterativo. Em geral, os graus dos polinômios que queremos multiplicar vão ser diferentes. Analisemos um caso particular de polinômios com graus diferentes: polinômios com graus supercrescentes.

Consideremos n polinômios p_i , tal que $\partial p_i = m_i$ e:

$$m_i > \sum_{j=1}^{i-1} m_j \quad 2 \leq i \leq n$$

Primeiro usemos uma estratégia tipo divisão e conquista. Suponhamos $n = 2^k$. A ordem dos produtos dos polinômios é:

$$(p_1 * p_2), (p_3 * p_4), \dots, (p_{n-1} * p_n)$$

Depois:

$$(p_1 p_2 * p_3 p_4), \dots, (p_{n-3} p_{n-2} * p_{n-1} p_n)$$

e assim sucessivamente, até calcular:

$$(p_1 p_2 \cdots p_{n/2}) * (p_{n/2+1} p_{n/2+2} \cdots p_n)$$

O custo deste algoritmo $\mathcal{A}$ é:

$$\begin{aligned} D(\mathcal{A}) &= \left(\sum_{i=1}^2 m_i + 1\right) + \left(\sum_{i=3}^4 m_i + 1\right) + \cdots + \left(\sum_{i=n-1}^n m_i + 1\right) + \\ &\quad + \sum_{i=1}^4 m_i + \cdots + \sum_{i=n-3}^n m_i + \cdots + \sum_{i=1}^n m_i \\ &= \left(\sum_{i=1}^n m_i + \frac{n}{2}\right) + \left(\sum_{i=1}^n m_i + \frac{n}{4}\right) + \cdots + \left(\sum_{i=1}^n m_i + 2\right) + \left(\sum_{i=1}^n m_i + 1\right) \\ &= k \sum_{i=1}^n m_i + \sum_{i=0}^{k-1} 2^i \\ &= \log n \sum_{i=1}^n m_i + (n - 1) \end{aligned}$$

Assintoticamente o algoritmo $\mathcal{A}$ tem ordem: $O(\log n \sum_{i=1}^n m_i)$. $\square$

Se multiplicarmos os polinômios segundo a ordem crescente dos graus, temos um algoritmo $\mathcal{B}$ de tipo tail recursion cuja ordem de multiplicação seria:

$$(\cdots ((p_1 * p_2) * p_3) \cdots * p_{n-1}) * p_n$$

Vejamos a complexidade do algoritmo $\mathcal{B}$.

$$D(\mathcal{B}) = \left(\sum_{i=1}^2 m_i + 1\right) + \left(\sum_{i=1}^3 m_i + 1\right) + \cdots + \left(\sum_{i=1}^{n-1} m_i + 1\right) + \left(\sum_{i=1}^n m_i + 1\right)$$

$$\begin{aligned}
&= \sum_{i=1}^2 m_i + \sum_{i=1}^3 m_i + \cdots + \sum_{i=1}^{n-1} m_i + \sum_{i=1}^n m_i + (n-1) \\
&< m_3 + m_4 + \cdots + m_n + \sum_{i=1}^n m_i + (n-1) \\
&< 2 \sum_{i=1}^n m_i + (n-1)
\end{aligned}$$

Do corolário 2.3 e do resultado anterior, concluimos que:

$$\sum_{i=1}^n m_i + 1 \leq D(P_m^n) \leq 2 \sum_{i=1}^n m_i + (n-1)$$

Podemos supor que os polinômios não são constantes, então:

$$n-1 < \sum_{i=1}^n m_i$$

e finalmente:

$$2 \sum_{i=1}^n m_i + (n-1) = O\left(\sum_{i=1}^n m_i\right)$$

Observações:

1) O resultado anterior significa que um algoritmo tipo tail recursion para polinômios com graus supercrescentes, usando uma ordem fixa de multiplicações, atinge em ordem a cota inferior do problema. Isto quer dizer que o problema de multiplicar vários polinômios não só tem a cota inferior dada pelo corolário 2.3, senão que tem essa complexidade para um determinado tipo de dados.

2) Como a condição de que os graus sejam supercrescentes é muito restritiva, vamos relaxar esta condição.

Intuitivamente, como a complexidade de um produto é a soma dos graus dos polinômios mais um, é melhor deixar para o último produto aquele polinômio que tem grau maior. Desta maneira, o grau desse polinômio vai aparecer menos vezes na conta da complexidade, obtendo um algoritmo melhor. Na próxima seção veremos que essa intuição é correta.

Suponha então que dentre os n polinômios a serem multiplicados existe um cujo grau é maior que a soma dos graus de todos os outros polinômios. Sem perda de generalidade, seja p_n esse polinômio. Então:

$$m_n > \sum_{i=1}^{n-1} m_i$$

Considere o seguinte algoritmo:

- i) multiplique de qualquer forma os $n - 1$ polinômios de grau menor
- ii) multiplique p_n pelo resultado do passo i)

Para determinar a complexidade do algoritmo, temos que m_n só aparece no último produto, e que:

$$\sum_{i=1}^{n-1} m_i = O(m_n)$$

Portanto o algoritmo tem ordem: $O(m_n)$, como no caso dos graus verificarem ser supercrescentes.

3) Se os graus dos polinômios a serem multiplicados são iguais, a melhor estratégia é divisão e conquista; se os graus são supercrescentes o melhor método é tail recursion. Não há uma estratégia que seja melhor que a outra, devendo-se considerar cada caso em especial.

4.1.3 Produto de polinômios com graus quaisquer

Sejam os polinômios $p_i : 1 \leq i \leq n$, tal que $\partial p_i = m_i$, onde os m_i são quaisquer. Os métodos aplicados para os casos especiais de mesmos graus ou graus supercrescentes caracterizam-se por fazer os produtos em ordem crescente de graus. Essa característica corresponde à idéia intuitiva de que, como a complexidade do produto é a soma dos graus mais um, é melhor multiplicar por último os polinômios com grau maior. Vamos provar que a melhor forma de multiplicar n polinômios, iterando o algoritmo para o produto de dois polinômios, é em ordem crescente de graus.

Construamos uma árvore binária de produtos aplicando um método guloso. A cada passo do produto de n polinômios, escolhe-se o par de polinômios com graus menores e multiplicam-se os dois polinômios aplicando o algoritmo ótimo para o produto de dois polinômios já citado. Cada folha da árvore representa um polinômio, e cada nó interno o polinômio resultante da operação de produto dos dois polinômios representados pelos seus filhos. Para cada vértice da árvore temos a informação do grau do polinômio que ele representa. Como o produto de dois polinômios de graus m_1 e m_2 tem grau $m_1 + m_2$ e tem complexidade $m_1 + m_2 + 1$, para cálculo da complexidade do produto só devemos somar 1 por cada nó interno. Cada grau vai contar em tantos nós quanto seja a profundidade do polinômio a que está associado. Se chamamos q_i à profundidade do polinômio p_i com grau m_i , $1 \leq i \leq n$, como no total são $n - 1$ nós internos, temos que a complexidade do produto dos n polinômios é:

$$\sum_{i=1}^n q_i m_i + (n - 1)$$

Chamamos *custo da árvore de produtos* a:

$$\sum_{i=1}^n q_i m_i$$

Um produto de polinômios 2 a 2 ótimo corresponde a uma árvore binária de produtos com custo mínimo, isto é, ótima.

Algoritmo : Constroi-Arvore-Produto

Este algoritmo constroi uma árvore binária de produto de polinômios que representa uma seqüência ótima de produtos de polinômios 2 a 2.

A entrada do algoritmo é um par (P, n) , onde P é um conjunto cujos elementos representam os polinômios a serem multiplicados, e n é a quantidade de elementos de P .

A saída do algoritmo é a raiz da árvore r .

O conjunto P é representado por um heap, no qual temos as operações de extração de mínimo, *ExMin*, e inserção de um elemento no conjunto P , *Ins*. O elemento remanescente do conjunto P , depois de aplicar o algoritmo Constroi-Arvore-Produto, é a raiz da árvore binária que representa o produto dos n polinômios.

Damos abaixo o corpo do algoritmo.

```
para i := 1 ate n-1 faça
    "obtenha um novo elemento apontado por N";
    N.esq := ExMin(P);
    N.dir := ExMin(P);
    Grau(N) := Grau(N.esq) + Grau(N.dir);
    Ins(N,P);
```

Teorema 4.2 *Seja P um conjunto que contém os valores $m_1, \dots, m_n$. O algoritmo Constroi-Arvore-Produto constroi corretamente uma árvore ótima de produtos correspondente à seqüência ótima de produtos de polinômios 2 a 2, com graus m_i : $1 \leq i \leq n$.*

Demonstração .- Por indução em n .

Base: para $n = 1$ a árvore construída pelo algoritmo contém um único vértice com o valor m_1 , e ela é obviamente ótima.

Passo: a hipótese indutiva é que o algoritmo Constroi-Árvore-Produto constroi uma árvore ótima de produtos 2 a 2 para todos os valores $m_1, \dots, m_l$ com $1 \leq l < n$.

Queremos provar que acontece o mesmo para todo conjunto $\{m_1, \dots, m_n\}$. Sem perda de generalidade podemos supor que:

$$m_1 \leq m_2 \leq \dots \leq m_n$$

Portanto na primeira iteração do comando "para" do algoritmo, os dois vértices determinados por ExMin contém os valores m_1 e m_2 . Ainda nesta primeira iteração, a subárvore A abaixo é construída:

Seja A' uma árvore ótima de produto de polinômios 2 a 2 para os valores $\{m_1, \dots, m_n\}$, e seja $w \in vA'$ um vértice interno de maior profundidade em A' . Se os filhos de w , m_i e m_j , não são m_1 e m_2 , podemos trocar dentro da árvore A' os valores $\{m_i, m_j\}$ pelos valores $\{m_1, m_2\}$ sem aumentar o custo da árvore. Concluimos que A é uma subárvore numa árvore ótima de produtos 2 a 2.

Ainda dentro de A' , substituímos A por uma folha contendo:

$$m_1 + m_2$$

A árvore resultante, A'' , é uma árvore ótima de produtos para os valores:

$$\{m_1 + m_2, m_3, \dots, m_n\}$$

Esta substituição corresponde à execução da primeira iteração do comando "para". Pela hipótese de indução, o algoritmo Constroi-Arvore-Produto constroi corretamente uma árvore ótima de produto para os valores:

$$\{m_1 + m_2, m_3, \dots, m_n\}$$

Portanto o algoritmo constroi uma árvore ótima de produtos para os valores $\{m_1, \dots, m_n\}$, correspondente ao produto de n polinômios 2 a 2, de forma ótima. $\square$

Corolário 4.2 *Se $\partial p_i = m : 1 \leq i \leq n$, o algoritmo Constroi-Arvore-Produto dá uma árvore balanceada, sendo que se $n = 2^k$ para algum k , ela é completa. O produto dos n polinômios é feito usando uma estratégia de divisão e conquista.*

Corolário 4.3 *Se $\partial p_i = m_i : 1 \leq i \leq n$ e:*

$$m_i > \sum_{j=1}^{i-1} m_j \quad 2 \leq i \leq n$$

o algoritmo Constroi-Arvore-Produto dá uma árvore totalmente desbalanceada. Tail recursion é o método usado para o produto dos polinômios.

A técnica desenvolvida no algoritmo Constroi-Arvore-Produto foi originalmente utilizada por Huffman [30] para achar um código ótimo que verifique a seguinte propriedade: dado um conjunto de caracteres e suas respectivas probabilidades de acontecer numa mensagem, nenhum código para um caractere é prefixo de nenhum código para outro caractere.

Esta propriedade do prefixo permite decodificar facilmente uma mensagem codificada eliminando prefixos da mesma que correspondem a codificações

de caracteres. O algoritmo de Huffman usa uma floresta onde cada árvore tem suas folhas rotuladas com os caracteres, as arestas são 0 ou 1 e a raiz é rotulada com a soma das probabilidades dos rótulos das folhas. No começo cada árvore contém um caractere só, e no final, o algoritmo dá uma só árvore com os caracteres como folhas. Nesta árvore o caminho da raiz para uma folha qualquer é o código para o rótulo desta folha, escrito com zeros e uns. O algoritmo usa um método guloso para juntar as árvores da floresta, escolhendo sempre as duas cujas raízes têm rótulo menor. Estudos detalhados do algoritmo de Huffman e uma implementação seguindo as diretivas acima, estão em Aho et alli [2] e Parker [44].

A idéia de usar um algoritmo guloso para obter uma árvore rotulada de comprimento mínimo a partir de uma floresta, também foi usada para demonstrar que o *mergesort* é um algoritmo que intercala listas ordenadas de forma ótima (Aho et alli [1], Terada [58]).

Nossa aplicação da técnica de Huffman visava a demonstração de que a melhor forma de multiplicar n polinômios é em ordem crescente de graus. Aparentemente, além do anterior, provamos que o método guloso do algoritmo Constroi-Arvore-Produto é ótimo; porém, se analisamos cuidadosamente, vemos que o algoritmo usa um heap para manter os graus em ordem crescente. A ordem para manter uma fila de prioridades (inserir um elemento e extrair o mínimo) é $O(n \log n)$, onde n é o tamanho do heap. Só que essa complexidade é medida segundo o número de *comparações* e não segundo o número de produtos !! (observe que manter os graus em ordem crescente é completamente necessário, e que a melhor forma de fazê-lo é usando uma fila de prioridades).

A única conclusão possível é que o algoritmo Constroi-Arvore-Produto deve ser considerado para provar o teorema anterior e não como um método ótimo segundo a nossa complexidade multiplicativa.

4.1.4 Problemas de implementação

Os resultados das seções anteriores são teóricos, e quando implementados na prática apresentam alguns problemas. Damos a seguir uma lista dos principais problemas junto com possíveis soluções.

1) Como o algoritmo para o produto dos polinômios A e B aplica a interpolação de Lagrange, os coeficientes do produto são calculados mediante a seguinte fórmula:

$$A(x)B(x) = \sum_{i=0}^{\partial A + \partial B} A(g_i)B(g_i) \frac{\prod_{j \neq i} (x - g_j)}{\prod_{j \neq i} (g_i - g_j)}$$

Temos que em cada somando, aparece uma divisão por:

$$\prod_{j \neq i} (g_i - g_j)$$

que não é considerada no nosso cálculo de complexidade, porém na prática não é irrelevante.

2) Se um polinômio é fixo, por exemplo A , podemos pré-computar:

$$\frac{A(g_i)}{\prod_{j \neq i} (g_i - g_j)} \quad 0 \leq i \leq \partial A + \partial B$$

multiplicando pelos $B(g_i)$, variando B , economizando operações aritméticas sem afetar a complexidade do algoritmo.

3) Suponhamos que queremos multiplicar 6 polinômios de grau 10 (não é exagerado!). Devemos considerar 61 constantes. Como o resultado do produto é um polinômio de grau 60, em algum momento vamos ter um produto cujos fatores têm graus com soma 60. Para exemplificar, digamos que os

dois polinômios têm grau 30. Na avaliação destes polinômios, até para constantes baixas, a capacidade da maioria dos computadores é excedida. Por exemplo, se a constante for 2, seria necessário calcular 2^{30} ; porém, 2^{30} é um número muito grande para ser armazenado como inteiro, e dá problemas de arredondamento usando ponto flutuante. Deve-se observar que 2 não é uma constante muito esquisita; pelo contrário é suficientemente simples para indicar que este problema é sério (quase toda constante excederia a capacidade do computador ou ocasionaria problemas de arredondamento).

Duas possíveis heurísticas são dadas a seguir:

i) Tomar $K = \mathbb{Z}_p$ com p primo. As constantes em $\mathbb{Z}_p$ asseguram-nos potências limitadas, pois só temos p elementos no corpo. Como $\text{car}(\mathbb{Z}_p) = p$, deve-se ter em conta na escolha do p , que seja maior que o número de constantes necessárias. Antes, como estávamos supondo que K era um corpo de característica zero, não tínhamos esse problema.

ii) Tomar as constantes no círculo unidade. Como as potências de uma raiz no círculo unidade são raízes do círculo, as constantes têm crescimento limitado. O número de raízes, no círculo unidade, necessárias é:

$$\sum_{i=1}^n m_i + 1$$

Seja m o número anterior. Como as m raízes no círculo unidade são números complexos, e estamos implementando os algoritmos no computador, não podemos avaliar os polinômios nesses pontos. Para contornar este problema, se g_i $i = 1, \dots, m$ são as raízes, tomaremos os polinômios módulo o produto:

$$\prod_{i=1}^m (x - g_i)$$

A idéia é aplicar o teorema chinês do resto, considerando o produto dos polinômios módulo os polinômios quadráticos que surgem de multiplicar:

$$(x - g_i) * (x - g_j)$$

onde g_j é a raiz conjugada de g_i . Desta maneira nos asseguramos que os termos sejam tratáveis pelo computador. Se o número de constantes for ímpar, pode-se considerar a constante zero, que também não cresce em magnitude ao computar as suas potências. Esta heurística dá uma solução pior quanto à complexidade, mas não usa constantes grandes (para algoritmos para o produto de polinômios módulo um polinômio e suas complexidades, veja Winograd [67] e Averbuch et al [4]).

4) Tendo em vista a definição de complexidade estabelecida, o número de adições do algoritmo não conta na sua complexidade. Porém, ao implementar algoritmos num computador, é bom reduzir esse número sem afetar a complexidade. Uma forma de fazer isso é escolher as constantes adequadamente. Como a quantidade de constantes consideradas é a mesma, a complexidade do algoritmo não é modificada.

5) Como já foi dito, nosso modelo de computação não considera o controle das recursões e a memória usada nas chamadas recursivas. Na prática, é bom levar em conta esses itens.

4.2 Polinômios em Várias Variáveis

Dividimos o estudo dos polinômios em várias variáveis em duas partes: polinômios densos e polinômios esparsos. Antes de analisarmos esses dois casos, mostraremos que o polinômio interpolador existe e é único. Continuamos assumindo K um corpo de característica zero. Restringiremos a demonstração para duas variáveis.

Teorema 4.3 *Sejam: $\{x_i : 0 \leq i \leq m, \text{ com } x_{i_1} \neq x_{i_2} \text{ se } i_1 \neq i_2\}$, $\{y_j : 0 \leq j \leq n, \text{ com } y_{j_1} \neq y_{j_2} \text{ se } j_1 \neq j_2\}$ e $\{p_{ij} : 0 \leq i \leq m, 0 \leq j \leq n\}$. Então existe um único polinômio da forma:*

$$p(x, y) = \sum_{k=0}^m \sum_{l=0}^n c_{kl} x^k y^l$$

que verifica: $p(x_i, y_j) = p_{ij} \quad \forall i, j$.

Demonstração .- Chamando:

$$\begin{aligned} L_k(x) &= \frac{\prod_{i \neq k} (x - x_i)}{\prod_{i \neq k} (x_k - x_i)} \\ L_l(y) &= \frac{\prod_{j \neq l} (y - y_j)}{\prod_{j \neq l} (y_l - y_j)} \\ L_{k,l}(x, y) &= L_k(x) \cdot L_l(y) \end{aligned}$$

como:

$$L_{k,l}(x_i, y_j) = \begin{cases} 1 & \text{se } k = i \text{ e } j = l \\ 0 & \text{outros casos} \end{cases}$$

temos que:

$$\boxed{p(x, y) = \sum_{k=0}^m \sum_{l=0}^n p(x_k, y_l) L_{k,l}(x, y)}$$

Portanto existe um polinômio em duas variáveis, verificando: $p(x_i, y_j) = p_{ij}$. A unicidade segue do fato do conjunto: $\{L_{kl} : 0 \leq k \leq m, 0 \leq l \leq n\}$ ser uma base do espaço dos polinômios em duas variáveis x, y , com potências máximas m e n , respectivamente. $\square$

4.2.1 Polinômios em várias variáveis densos

Consideremos o produto de dois polinômios em duas variáveis x e y , com potências máximas m e n , respectivamente. A idéia central do algoritmo a considerar é interpolar primeiro numa variável, e o resultado obtido ser interpolado na outra variável. Nesta seção provaremos que esta idéia simples leva a uma solução ótima.

Sejam:

$$p(x, y) = \sum_{k=0}^m \sum_{l=0}^n a_{kl} x^k y^l \quad e \quad q(x, y) = \sum_{k=0}^m \sum_{l=0}^n b_{kl} x^k y^l$$

Seu produto é:

$$p(x, y)q(x, y) = \sum_{k=0}^{2m} \sum_{l=0}^{2n} c_{kl} x^k y^l$$

onde:

$$c_{kl} = \sum_{i=0}^k \sum_{j=0}^l a_{ij} b_{k-i, l-j}$$

O problema é calcular os c_{kl} sobre a base $\{a_{kl}\} \cup \{b_{kl}\}$.

Escrevamos o produto de dois polinômios de acordo com a expressão do polinômio interpolador do teorema 4.3.

$$p(x, y)q(x, y) = \sum_{k=0}^{2m} \sum_{l=0}^{2n} p(x_k, y_l) q(x_k, y_l) L_k(x) L_l(y)$$

Se:

$$L_k(x) = \sum_{i=0}^{2m} L_{ki} x^i$$

$$L_l(y) = \sum_{j=0}^{2n} L_{lj} y^j$$

temos que:

$$\begin{aligned}
 p(x, y)q(x, y) &= \sum_{k=0}^{2m} \sum_{l=0}^{2n} p(x_k, y_l) q(x_k, y_l) \left(\sum_{i=0}^{2m} L_{ki} x^i \right) \left(\sum_{j=0}^{2n} L_{lj} y^j \right) \\
 &= \sum_{k=0}^{2m} \sum_{l=0}^{2n} \left(\sum_{i=0}^{2m} \sum_{j=0}^{2n} p(x_k, y_l) q(x_k, y_l) L_{ki} L_{lj} x^i y^j \right) \\
 &= \sum_{i=0}^{2m} \sum_{j=0}^{2n} \left(\sum_{k=0}^{2m} \sum_{l=0}^{2n} p(x_k, y_l) q(x_k, y_l) L_{ki} L_{lj} \right) x^i y^j
 \end{aligned}$$

Então os coeficientes do polinômio produto são:

$$c_{ij} = \sum_{k=0}^{2m} \sum_{l=0}^{2n} p(x_k, y_l) q(x_k, y_l) L_{ki} L_{lj}$$

com $0 \leq i \leq 2m$ e $0 \leq j \leq 2n$.

Algoritmo para o produto de 2 polinômios em duas variáveis

Sejam $x_0, x_1, \dots, x_{2m}$ constantes diferentes, e $y_0, y_1, \dots, y_{2n}$ também constantes diferentes entre si.

i) Calcule: $p(x_k, y_l) q(x_k, y_l)$ $k = 0, \dots, 2m; l = 0, \dots, 2n$

ii) Considere:

$$\begin{aligned}
 L_k(x) &= \frac{\prod_{i \neq k} (x - x_i)}{\prod_{i \neq k} (x_k - x_i)} = \sum_{i=0}^{2m} L_{ki} x^i \\
 L_l(y) &= \frac{\prod_{j \neq l} (y - y_j)}{\prod_{j \neq l} (y_l - y_j)} = \sum_{j=0}^{2n} L_{lj} y^j
 \end{aligned}$$

iii) Finalmente :

$$c_{ij} = \sum_{k=0}^{2m} \sum_{l=0}^{2n} p(x_k, y_l) q(x_k, y_l) L_{ki} L_{lj}$$

para $0 \leq i \leq 2m$ e $0 \leq j \leq 2n$.

Provemos que o algoritmo é ótimo.

Computar $p(x_k, y_l)$ e $q(x_k, y_l)$ tem custo zero, pois $x_k, y_l \in K$. Para cada k e cada l , o custo de computar $p(x_k, y_l) \cdot q(x_k, y_l)$ é 1. Como os coeficientes de $L_k(x)$, L_{ki} , e de $L_l(y)$, L_{lj} , são elementos de K , podemos computar cada coeficiente do produto com: $(2m + 1)(2n + 1)$ multiplicações. Como todos os coeficientes utilizam os mesmos produtos essenciais, temos que a complexidade do algoritmo é: $(2m + 1)(2n + 1)$. Usando o corolário 2.4 para duas variáveis, concluímos que o algoritmo é ótimo. $\square$

No caso do produto de 2 polinômios em várias variáveis, o algoritmo ótimo usado para duas variáveis pode ser aqui aplicado, obtendo-se também otimalidade (mesmo que as indeterminadas dos polinômios sejam diferentes, bem como suas respectivas potências máximas).

Para o caso de produtos de n polinômios em várias variáveis, deve-se fazer um estudo similar ao feito para polinômios numa variável. Ou seja, dado o custo de multiplicar dois polinômios em várias variáveis, aplicar o gerador de algoritmos multilineares quase-ótimos obtendo algoritmos para o produto dos n polinômios. Observe que o custo de multiplicar 2 polinômios depende das potências máximas em cada indeterminada, devendo-se analisar os diferentes casos possíveis.

Em respeito à paciência do leitor que nos acompanhou até aqui e “a la premura del tiempo”, não nos deteremos em tais análises.

4.2.2 Polinômios esparsos

Consideremos polinômios em várias variáveis esparsos. Apesar de na prática serem mais importantes que os polinômios densos, até recentemente não eram conhecidos algoritmos para interpolar tais polinômios de forma determinística e em tempo polinomial (algoritmos probabilísticos foram estudados por Zippel [68] [69]).

No ano de 1988, Ben-Or e Tiwari [5] publicaram um algoritmo determinístico para interpolar polinômios esparsos, baseado na decodificação de códigos BCH. O modelo de computação usado foi o RAM (Random Access Machines), no qual todas as operações aritméticas (+, −, ×, ÷) têm o mesmo custo (para um estudo do modelo RAM, veja Aho et alli [1]).

O algoritmo de Ben-Or e Tiwari tem uma complexidade da $O(nd\tau^3 \log n)$, onde:

n - número de variáveis

d - grau total do polinômio

τ - cota superior do número de termos não nulos do polinômio

A seguir damos o algoritmo de Ben-Or e Tiwari, medindo a sua complexidade de segundo nosso modelo. Vamos supor que conhecemos o número exato de termos não nulos, t , do polinômio a interpolar.

Aplicações de um algoritmo de interpolação são: produto de polinômios esparsos em várias variáveis e fatoração de polinômios (Kaltofen e Trager [32]).

Considerações iniciais

Seja:

$$P(x_1, \dots, x_n) = c_1 m_1 + \dots + c_t m_t$$

o polinômio a ser interpolado. Os coeficientes c_j são todos diferentes de zero; os monômios:

$$m_j = x_1^{e_{j,1}} x_2^{e_{j,2}} \dots x_n^{e_{j,n}}$$

são diferentes dois a dois, e t é o número de termos em P .
Seja p_j o j -ésimo primo, e:

$$v_j = p_1^{e_{j,1}} p_2^{e_{j,2}} \dots p_n^{e_{j,n}}$$

a avaliação do monômio m_j em $(p_1, \dots, p_n)$. Com esta avaliação $m_i \neq m_j$ se e só se $v_i \neq v_j$.

Chamemos:

$$a_i = P(p_1^i, \dots, p_n^i) = \sum_{j=1}^t c_j v_j^i \quad 0 \leq i \leq 2(t-1)$$

Primeiramente, o algoritmo calcula os v_j e depois obtém os coeficientes, c_j , do polinômio. Faremos uso do polinômio auxiliar:

$$q(z) = \prod_{j=1}^t (z - v_j) = z^t + q_{t-1}z^{t-1} + \dots + q_0$$

Como:

$$\sum_{j=1}^t c_j v_j^i q(v_j) = \sum_{k=0}^{t-1} q_k (c_1 v_1^{k+i} + c_2 v_2^{k+i} + \dots + c_t v_t^{k+i}) + (c_1 v_1^{t+i} + c_2 v_2^{t+i} + \dots + c_t v_t^{t+i})$$

para $0 \leq i \leq t-1$ e $q(v_j) = 0$, temos o sistema de equações de Toeplitz, $T_{t-1,t-1} q_{t-1} = -a_{2t-1,t-1}$ seguinte:

$$\begin{bmatrix} a_{t-1} & a_t & \dots & a_{2t-2} \\ a_{t-2} & a_{t-1} & \dots & a_{2t-3} \\ \vdots & \vdots & \ddots & \vdots \\ a_0 & a_1 & \dots & a_{t-1} \end{bmatrix} \begin{pmatrix} q_0 \\ q_1 \\ \vdots \\ q_{t-1} \end{pmatrix} = - \begin{pmatrix} a_{2t-1} \\ a_{2t-2} \\ \vdots \\ a_t \end{pmatrix}$$

Podemos escrever a matriz do sistema, $T_{t-1,t-1}$, como:

$$T_{t-1,t-1} = \begin{bmatrix} v_1^{t-1} & v_2^{t-1} & \dots & v_t^{t-1} \\ v_1^{t-2} & v_2^{t-2} & \dots & v_t^{t-2} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \dots & 1 \end{bmatrix} \begin{bmatrix} c_1 & 0 & \dots & 0 \\ 0 & c_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & c_t \end{bmatrix} \begin{bmatrix} 1 & v_1 & \dots & v_1^{t-1} \\ 1 & v_2 & \dots & v_2^{t-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & v_t & \dots & v_t^{t-1} \end{bmatrix}$$

Como os v_j são distintos, as matrizes de Vandermonde são inversíveis, e como os c_j são diferentes de zero, o sistema é não singular. As raízes do polinômio $q(z)$ dão os v_j . Tomando a avaliação do polinômio P nos pontos: $(p_1^i, \dots, p_n^i)$ para $0 \leq i \leq t-1$, obtemos o sistema de Vandermonde transposto, $Ac = a$, seguinte:

$$\begin{bmatrix} 1 & 1 & \dots & 1 \\ v_1 & v_2 & \dots & v_t \\ \vdots & \vdots & \ddots & \vdots \\ v_1^{t-1} & v_2^{t-1} & \dots & v_t^{t-1} \end{bmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_t \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{t-1} \end{pmatrix}$$

cuja solução são os coeficientes do polinômio P .

Algoritmo para interpolar polinômios esparsos

Entrada: uma caixa preta para avaliar um polinômio em várias variáveis $P(x_1, \dots, x_n)$ e o número exato de termos não nulos do polinômio.

Saída: O polinômio $P = \sum_{j=1}^t c_j m_j$.

(1) Avaliar P em $(p_1^i, \dots, p_n^i)$ para $0 \leq i \leq 2t-2$. Sejam $a_0, \dots, a_{2t-2}$ os valores respectivos.

(2) Resolver o sistema de Toeplitz: $T_{t-1,t-1} q_{t-1} = -a_{2t-1,t-1}$ obtendo: $q(z) = \sum_{j=1}^t q_j z^j$.

(3) Achar as raízes inteiras de $q(z)$ obtendo os v_j . Dividindo cada v_j sucessivamente por $p_1, \dots, p_n$, obtemos os monômios m_j .

(4) Resolver o sistema de Vandermonde transposto $Ac = a$, especificado acima, achando os coeficientes c_j do polinômio P .

Analisemos os diferentes passos do algoritmo de Ben-Or e Tiwari, calculando suas respectivas complexidades.

Resolução de Sistemas de Toeplitz

No passo (2), temos que resolver o sistema, $T_{t-1,t-1} q_{t-1} = -a_{2t-1,t-1}$ onde T é uma matriz de Toeplitz não singular. A seguir consideraremos o algoritmo de Brent et alli [8] para resolver um sistema de Toeplitz. Para simplificar a notação, consideremos a matriz do sistema de dimensões $(n+1) \times (n+1)$. Sejam: $x = (x_0, \dots, x_n)$ e $y^r = (y_n, \dots, y_0)$, a primeira coluna e a primeira linha, respectivamente, da matriz inversa de T . Se $x_0 = y_0 \neq 0$, a matriz inversa de T (Brent et alli [8], teorema 6) é:

$$T^{-1} = \frac{1}{x_0} \left(\begin{bmatrix} x_0 & 0 & \dots & 0 \\ x_1 & x_0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ x_n & x_{n-1} & \dots & x_0 \end{bmatrix} \begin{bmatrix} y_0 & y_1 & \dots & y_n \\ 0 & y_0 & \dots & y_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & y_0 \end{bmatrix} - \begin{bmatrix} 0 & 0 & \dots & 0 \\ y_n & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ y_1 & y_2 & \dots & 0 \end{bmatrix} \begin{bmatrix} 0 & x_n & \dots & x_1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & x_n \\ 0 & 0 & \dots & 0 \end{bmatrix} \right) \quad (4.1)$$

Brent et alli estendem a fórmula fechada anterior ao caso em que $x_0 = 0$ e $\det(T) \neq 0$ (teorema 7 do artigo citado). A idéia central do teorema é estender a matriz T a uma matriz $T'_{(n+2) \times (n+2)}$, de modo tal que se

$x' = (x'_0, \dots, x'_{n+1})$ e $(y')^r = (y'_{n+1}, \dots, y'_0)$ são a primeira coluna e a primeira linha, respectivamente, de $(T')^{-1}$, então $x'_0 = y'_0 \neq 0$ e:

$$T^{-1} = \frac{1}{x'_0} \left(\begin{bmatrix} x'_0 & 0 & \dots & 0 \\ x'_1 & x'_0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ x'_n & x'_{n-1} & \dots & x'_0 \end{bmatrix} \begin{bmatrix} y'_0 & y'_1 & \dots & y'_n \\ 0 & y'_0 & \dots & y'_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & y'_0 \end{bmatrix} - \begin{bmatrix} y'_{n+1} & 0 & \dots & 0 \\ y'_n & y'_{n+1} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ y'_1 & y'_2 & \dots & y'_{n+1} \end{bmatrix} \begin{bmatrix} x'_{n+1} & x'_n & \dots & x'_1 \\ 0 & x'_{n+1} & \dots & x'_2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & x'_{n+1} \end{bmatrix} \right) \quad (4.2)$$

Então temos que: $q_{t-1} = -T_{t-1,t-1}^{-1} a_{2t-1,t-1}$. Usando 4.1 ou 4.2 para obter q_{t-1} temos que fazer quatro produtos de matrizes de Toeplitz por vetores, e uma soma. Os produtos envolvidos no cálculo de q_{t-1} , podem ser vistos como produto de polinômios. Por exemplo:

$$\begin{bmatrix} y_0 & y_1 & \dots & y_n \\ 0 & y_0 & \dots & y_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & y_0 \end{bmatrix} \begin{pmatrix} a_{2t-1} \\ a_{2t-2} \\ \vdots \\ a_t \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{t-1} \end{pmatrix}$$

corresponde a $B(x) = (Y(x) Q^r(x)) \bmod x^t$ onde: $B(x) = \sum_{i=0}^{t-1} b_i x^i$, $Y(x) = \sum_{i=0}^{t-1} y_i x^i$ e $A(x) = \sum_{i=0}^{t-1} a_{t+i} x^i$.

A complexidade de resolver o sistema de Toeplitz, dadas a primeira coluna e a primeira linha de T^{-1} , está dada por 4 produtos de polinômios e uma soma de polinômios, e portanto é: $4(2t+1) = O(t)$.

Falta, então, calcular a primeira coluna, x , e a primeira linha, y , respectivamente, de T^{-1} ou $(T')^{-1}$. Para isso, Brent et alli observaram a relação existente entre resolver um sistema de Toeplitz e achar um tipo de aproximação por funções racionais de uma série de potências, chamada *aproximação*

de Padé. Isto é, se $A(x) = a_0 + a_1x + \dots$ é uma série de potências, a aproximação (m, p) de Padé de A é a função racional: $R_{mp} = U(x)/V(x)$ onde U e V são polinômios, com $\text{grau}(U) \leq m$ e $\text{grau}(V) \leq p$. O conjunto de aproximações de Padé de $A(x)$, variando m e p , dispostas numa matriz de duas dimensões é chamado de *tabela de Padé*. Consideremos o polinômio: $P(x) = a_0 + a_1x + \dots + a_{2n}x^{2n}$. A matriz de Toeplitz associada ao polinômio P é:

$$T = \begin{bmatrix} a_n & a_{n-1} & \dots & a_0 \\ a_{n+1} & a_n & \dots & a_1 \\ \vdots & \vdots & \ddots & \vdots \\ a_{2n} & a_{2n-1} & \dots & a_n \end{bmatrix}$$

A aproximação (n, n) de Padé do polinômio P dá a primeira coluna, x , de T^{-1} ; analogamente, considerando T^{tr} , a aproximação (n, n) de Padé do polinômio: $P^r(x) = a_{2n} + a_{2n-1}x + \dots + a_0x^{2n}$ dá a primeira linha, y , de T^{-1} (veja seção 6.a do artigo citado).

Então, para calcular a primeira coluna, x , e a primeira linha, y , de T^{-1} , temos que calcular as aproximações (n, n) de Padé associadas a T e a T^{tr} . Os algoritmos **AD** (diagonal secundária) e **MD** (diagonal principal), das seções 4 e 5 do artigo que estamos considerando, calculam a aproximação (n, n) de Padé do polinômio P , percorrendo a diagonal secundária e a diagonal principal, respectivamente, da tabela de Padé. O algoritmo AD baseia-se no algoritmo estendido de Euclides usando divisão e conquista, enquanto que o MD também usa divisão e conquista, obtendo a solução final através de um número fixo de produtos de polinômios adequados. Usando MD, o custo de achar a primeira coluna, x , e a primeira linha, y , de T^{-1} , verifica:

$$D(n) \leq D\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + D\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n)$$

onde n indica que estamos calculando a aproximação (n, n) de Padé, e $O(n)$ é o custo dos produtos de polinômios. Logo, o custo de achar x e y é $O(n \log n)$ para uma matriz de Toeplitz de dimensões $(n+1) \times (n+1)$. Aplicando o

resultado anterior ao caso $T_{t-1,t-1}$, temos: $O(t \log t)$.

Algoritmo para resolver um sistema de Toeplitz não singular

Entrada: os elementos da matriz de Toeplitz não singular, $T_{t-1,t-1}$: $a_0, \dots, a_{2t-2}$

Saída: $q_0, \dots, q_{t-1}$ solução de: $T_{t-1,t-1} q_{t-1} = -a_{2t-1,t-1}$.

(1) Calcular a primeira coluna, x , e a primeira linha, y , de T^{-1} usando o algoritmo MD (seção 6 do artigo de Brent [8]).

(2) Resolva o sistema de Toeplitz, não singular, aplicando as fórmulas 4.1 ou 4.2 para T^{-1} .

A complexidade do algoritmo é dominada pelo passo (1), e então o algoritmo é $O(t \log t)$.

Raízes inteiras de um polinômio

No passo (3) do algoritmo de interpolação temos que achar as raízes inteiras do polinômio $q(z) = \sum_{i=0}^{t-1} q_i z^i + z^t$. Aplicaremos um algoritmo tipo Newton-Hensel devido a Loos [35]. A idéia do algoritmo é mudar o domínio dos coeficientes do polinômio para Z_p , onde p é primo, fazer uma busca exaustiva nos p elementos de Z_p e voltar para o polinômio original com coeficientes em Z , conhecendo agora os zeros do polinômio com coeficientes em Z_p .

Consideremos um morfismo: $q^* = q \bmod p$ que leva os coeficientes de q em Z nos coeficientes de q^* em Z_p . O morfismo anterior deixa 0,1, somas e multiplicações invariantes. Como a operação de testar se α é raiz de q só depende de produtos e de adições, esse cálculo também fica invariante pelo morfismo. Isto significa que, se as raízes inteiras de q estão em Z_p , então elas formam um subconjunto das raízes inteiras de q^* , que podemos achar testando. Se as

raízes inteiras de q não estão em Z_p , resta voltar para Z , conhecendo os zeros de q^* em Z_p . Para isso levantamos (no sentido de Hensel) os zeros de q^* aos zeros de q segundo o seguinte teorema, que enunciamos sem demonstração (veja Loos [35]):

Teorema 4.4 *Seja $q(z)$ um polinômio sobre Z e α um zero de $q \bmod p$, que não é zero da derivada $q'(z) \equiv 0 \bmod p$, com p primo. Então para cada $r \geq 1$ a congruência $q(z) \equiv 0 \bmod p^r$ tem uma solução $\alpha_r \equiv \alpha \bmod p$.*

Na demonstração do teorema mostra-se como calcular as sucessivas expansões p -ádicas (levantamentos). No nosso caso, como $q(z) = \prod_{j=1}^t (z - v_j)$ e os v_j são diferentes, todas as hipóteses verificam-se, e temos:

$$\alpha_{i+1} = \alpha_i + (((q(\alpha_i)/p^i)/q'(\alpha_i)) \bmod p) p^i \quad (4.3)$$

O primo p tem que ser o menor, tal que $q \bmod p$ seja livre de quadrados (isto é, $q \bmod p$ e $q' \bmod p$ não têm zeros comuns). No teorema 2 do artigo citado, prova-se que o primo p é $O(t \log t)$, portanto é possível fazer uma busca exaustiva em Z_p .

Para completarmos o algoritmo devemos obter uma cota do número de levantamentos, N , para cada zero de q^* . Se procurássemos as raízes racionais de q , s/u , os inteiros $q_t s$ e $q_0 u$ teriam que estar entre: $-[p^{N/2}], \dots, [p^{N/2}]$, isto é, N é o menor inteiro tal que:

$$-[p^{N/2}] \leq |q_t| |q_0| < [p^{N/2}]$$

Como $q_t = 1$, N é o menor inteiro que verifica $2 | q_0 | < p^N$. Exibimos a seguir o algoritmo de Loos para achar raízes inteiras de polinômios (o algoritmo original de Loos encontra as raízes racionais do polinômio).

Algoritmo de Loos

Entrada: o polinômio $q(z) = \sum_{i=0}^{t-1} q_i z^i + z^t$ sobre os inteiros, livre de quadrados.

Saída: a lista L dos zeros inteiros de q .

- (1) Seja p o menor primo tal que $q \bmod p$ é livre de quadrados.
- (2) Seja $b = 2 \mid q_0 \mid$, e escolha o menor N tal que $b < p^N$.
- (3) Calcule a lista L_1 dos zeros de $q^* = q \bmod p$ por busca exaustiva.
- (4) Levante os zeros em L_1 de q^* , aos zeros de $q \bmod p^N$ usando a fórmula 4.3.
- (5) Para cada α em L_1 , se $q(\alpha) = 0$ inclua α em L .

Analisemos a complexidade do algoritmo. Para achar o primo p temos que calcular $q \bmod p$ e $q' \bmod p$ (custo $O(t)$), e calcular o $\text{mdc}(q \bmod p, q' \bmod p)$. O custo de se computar o máximo divisor comum de dois polinômios de grau t e $t - 1$, respectivamente, é $O(t \log t)$, usando o algoritmo HGCD (Aho et alli [1]). A complexidade de se testar se $q \bmod p$ é livre de quadrados para um número p em $\mathbb{Z}_p$ é $O(t \log t)$. Usando um algoritmo rápido de avaliação (veja o capítulo 4 da monografia de Borodin e Munro [7]), a complexidade do passo (1) é: $O(t \log t \log(t \log t))$.

Para determinar N , temos que multiplicar p por si mesmo enquanto $p_i < b$ e parar quando $p_N > b$. O número de produtos é $O(\log b)$, i.é., $O(\log B)$ onde B é uma cota superior dos coeficientes do polinômio q .

No passo (3) devem-se testar $O(t \log t)$ números e, analogamente ao passo (2), usando o método de Horner para avaliar o polinômio num ponto e algoritmos rápidos de avaliação tipo divisão e conquista temos: $O(t \log(t \log t))$. Para cada raiz α de q^* temos N levantamentos até $q \bmod p^N$. O cálculo de um levantamento é feito segundo a fórmula 4.3, e é dominada pela avaliação de $q(\alpha_i)$, portanto $O(t)$. No máximo são t raízes a serem levantadas, logo a complexidade do passo (4) é: $O(t^2 \log B)$, onde $\log B$ é uma cota superior para N .

O custo de se testar se α^* verifica $q(\alpha^*) = 0$ é $O(t)$. Como são t raízes, analogamente aos passos (2) e (3), temos no passo (5) $O(t \log t)$.

Finalmente, a complexidade do algoritmo de Loos, para achar as raízes inteiras do polinômio q é dominada pela do passo (4), sendo $O(t^2 \log B)$.

Como as raízes de $q(z)$ são os $v_j = p_1^{e_{j,1}} p_2^{e_{j,2}} \dots p_n^{e_{j,n}}$, podemos determinar os monômios $m_j = x_1^{e_{j,1}} x_2^{e_{j,2}} \dots x_n^{e_{j,n}}$, dividindo os v_j repetidamente por

$p_1, \dots, p_n$ com custo zero.

Concluimos que a complexidade do passo (3) do algoritmo para interpolar um polinômio em várias variáveis esparsas é: $O(t^2 \log B)$.

Resolução de Sistemas de Vandermonde Transpostos

No passo (4) do algoritmo de interpolação de um polinômio em várias variáveis temos que resolver o sistema: $Ac = a$, sendo A uma matriz de Vandermonde transposta. O algoritmo para resolver um sistema de Vandermonde transposto que damos abaixo é de Kaltfen e Yagati [31]. Seja $B = A^{tr}$ a matriz transposta de A . Então: $c = (B^{-1})^{tr} a$. Formemos para cada j , $1 \leq j \leq t$, o polinômio $B_j(z) = b_{0,j} + b_{1,j}z + \dots + b_{t-1,j}z^{t-1}$, a partir da j -ésima coluna de B^{-1} , $(b_{0,j}, b_{1,j}, \dots, b_{t-1,j})^{tr}$. A posição (i, j) de BB^{-1} é:

$$B_j(v_i) = \begin{cases} 1 & \text{se } i = j \\ 0 & \text{outros casos} \end{cases}$$

Então:

$$B_j(z) = \frac{\prod_{k \neq j} (z - v_k)}{\prod_{k \neq j} (v_j - v_k)} \quad 1 \leq j \leq t$$

e:

$$c = (B^{-1})^{tr} a = \begin{pmatrix} \sum_{i=0}^{t-1} b_{i,1} a_i \\ \sum_{i=0}^{t-1} b_{i,2} a_i \\ \vdots \\ \sum_{i=0}^{t-1} b_{i,t} a_i \end{pmatrix}$$

Por outro lado, chamemos: $D(z) = a_0 z^t + a_1 z^{t-1} + \dots + a_{t-1} z$.

O coeficiente de z^t em $B_i(z)D(z)$ é c_i . Uma forma mais rápida de calcular os c_i é considerando que:

$$B_i(z) = \frac{B(z)}{\alpha_i(z - v_i)}$$

onde: $B(z) = \prod_{j=1}^t (z - v_j)$ e $\alpha_i = \prod_{i \neq j} (v_i - v_j)$. Seja:

$$B(z)D(z) = q_{2t}z^{2t} + \dots + q_1z$$

O coeficiente de z^t em $B(z)D(z)/(z - w)$ é dado por:

$$Q_t(w) = q_{2t}w^{t-1} + q_{2t-1}w^{t-2} + \dots + q_{t+1}$$

Então:

$$Q_t(v_i) = \alpha_i c_i \quad 1 \leq i \leq t$$

Para obter os coeficientes c_i , falta calcular os α_i e finalmente: $(\alpha_i c_i)/\alpha_i$.
Visando calcular os α_i , observemos que:

$$B'(z) = \frac{dB(z)}{dz} = \sum_{i=1}^t \prod_{i \neq j} (z - v_j)$$

e então: $B'(v_i) = \alpha_i$ $1 \leq i \leq t$. O algoritmo que avalia $Q_t(w)$ em $v_1, \dots, v_t$, pode ser usado para avaliar $B'(z)$ nos mesmos pontos.

Algoritmo para resolver um sistema de Vandermonde transposto

Entrada: os elementos da matriz A de Vandermonde transposta, $v_1, \dots, v_t$ e o vetor $a = (a_0, \dots, a_{t-1})^{tr}$.

Saída: o vetor $c = (c_1, \dots, c_t)^{tr}$ $c = A^{-1}a$.

(1) Calcule o polinômio: $B(z) = \prod_{j=1}^t (z - v_j)$.

(2) Seja $D(z)$ como acima. Calcule: $B(z)D(z) = q_{2t}z^{2t} + \dots + q_1z$ e chame: $Q_t(w) = q_{2t}w^{t-1} + q_{2t-1}w^{t-2} + \dots + q_{t+1}$.

(3) Avalie $Q_t(w)$ nos pontos $v_1, \dots, v_t$ obtendo $\alpha_i c_i$.

(4) Calcule os α_i avaliando $B'(z)$ nos pontos $v_1, \dots, v_t$.

(5) Calcule os c_i dividindo $Q_t(v_i)/B'(v_i)$ $1 \leq i \leq t$.

Analisemos a complexidade do algoritmo. O primeiro passo tem complexidade $t - 1$. Multiplicar dois polinômios de grau t tem complexidade $2t + 1$. Para avaliar o polinômio $Q_t(w)$ nos pontos $v_1, \dots, v_t$ usaremos um algoritmo rápido, tipo divisão e conquista, sendo que a avaliação num ponto é feita segundo o método de Horner. Para isso, se:

$$Q_t(w) = (\dots((q_{2t}w + q_{2t-1})w + q_{2t-2})\dots)w + q_{t+1}$$

o custo de avaliar $Q_t(w)$ num ponto é: $t - 1$ e a complexidade total é $O(t \log t)$ (veja o capítulo 4 da monografia de Borodin e Munro [7]).

O passo (4) aplica o mesmo algoritmo do passo (3), e portanto sua complexidade é da $O(t \log t)$. Finalmente, o passo (5) são t divisões. A complexidade total do algoritmo para resolver um sistema de Vandermonde transposto é: $O(t \log t)$.

Conclusão

Devido a nosso modelo de computação, o passo (1) do algoritmo de interpolação para polinômios esparsos tem custo zero (é interessante ver que este resultado independe da estrutura de dados usada para armazenar o polinômio). Claramente o passo (3) é dominante na complexidade total do algoritmo, e portanto a sua complexidade é: $O(t^2 \log B)$, onde t é o número exato de coeficientes diferentes de zero em P , e B é uma cota superior nos coeficientes do polinômio q do passo (3).

Se não sabemos qual o número de coeficientes diferentes de zero em P , mas temos uma cota superior: $\tau > t$, podemos alterar o algoritmo de modo a

continuar sendo um método aplicável. Neste caso, o problema principal é que tendo como entrada $\tau > t$ avaliações, os coeficientes c_k para $k > t$ são zero, e então a matriz de Toeplitz $T_{\tau,\tau}$ é singular. O algoritmo de Brent usado para resolver o sistema de Toeplitz do passo (2): $T_{t-1,t-1} q_{t-1} = -a_{2t-1,t-1}$, exige que a matriz do sistema seja não singular, portanto não podemos usá-lo. A resolução de sistemas de Toeplitz gerais foi estudada por Kautsky e Yagati [31] quando o modelo de computação é RAM, mostrando um algoritmo da $O(M(\tau) \log \tau)$, onde $M(\tau)$ é o número total de operações aritméticas necessárias para multiplicar 2 polinômios de grau τ . Como tal algoritmo é straight line, $O(M(\tau) \log \tau)$ é uma cota superior no modelo de complexidade multiplicativa, já que este considera menos operações que o modelo RAM. Sendo $M(\tau) = \tau$ no nosso caso, temos que a alteração do algoritmo para resolver sistemas de Toeplitz singulares tem custo $O(\tau \log \tau)$. Portanto a adaptação do algoritmo de interpolação para polinômios esparsos ao caso em que o número exato de termos diferentes de zero não é conhecido tem complexidade: $O(\tau \log \tau + t^2 \log B)$.

Bibliografia

- [1] Aho, A.V. , Hopcroft, J.E. and Ullman, J.D.
The design and analysis of computer algorithms.
Addison-Wesley, Reading, Mass, 1974.
- [2] Aho, A.V. , Hopcroft, J.E. and Ullman, J.D.
Data structures and algorithms.
Addison-Wesley, Reading, Mass, 1983.
- [3] Alder, A. and Strassen, V.
On the algorithmic complexity of associative algebras.
Theoret. Comput. Sci., vol. 15, 1981, 201-211.
- [4] Averbuch, A. , Galil, Z. and Winograd, S.
*Classification of all the minimal bilinear algorithms for computing the
coefficients of the product of two polynomials modulo a polynomial.*
Part I : the algebra $G[u]/\langle Q(u)^l \rangle$ $l > 1$
Theoret. Comput. Sci., vol. 58, 1988, 17-56.
- [5] Ben-Or, M. and Tiwari, P.
A deterministic algorithm for sparse multivariate polynomial interpolation .
20th Annual ACM Symp. Theory Comp., 1988, 301-309.
- [6] Bini, D. , Capovani, M. , Lotti, G. and Romani, F.
 $O(n^{2.7799})$ complexity for approximate matrix multiplication.
Information Proc. Letters, vol. 8, 1979, 234-235.
- [7] Borodin, A. and Munro, I.
The computational complexity of algebraic and numeric problems.
American Elsevier , 1975.

- [8] Brent, R., Gustavson, F. and Yun, D.
Fast solution of Toeplitz systems of equations and computation of Padé approximants.
 J. of Algorithms, vol. 1, 1980, 259-295.
- [9] Brockett, R.W. and Dobkin, D.
On the number of multiplications required for matrix multiplication.
 SIAM J. Comput., vol. 5, 1976, 624-628.
- [10] Büchi, W. and Clausen, M.
On a class of primary algebras of minimal rank.
 Linear Algebra and its Appl., vol. 69, 1985, 249-268.
- [11] Bunch, J. and Hopcroft, J.E.
Triangular factorization and inversion by fast matrix multiplication.
 Math. Comp., vol. 28, 1974, 231-236.
- [12] Cooley, J.W. and Tukey, J.W.
An algorithm for the machine calculation of complex Fourier series.
 Math. Comp., vol. 19, 1965, 297-301.
- [13] Coppersmith, D.
Rapid multiplication of rectangular matrices.
 SIAM J. Comput., vol. 11, 1982, 467-471.
- [14] Coppersmith, D. and Winograd, S.
On the asymptotic complexity of matrix multiplication.
 SIAM J. Comput., vol. 11, 1982, 472-492.
- [15] De Groote, H.F.
On varieties of optimal algorithms for the complexity of bilinear mappings. I. The isotropy group of a bilinear mapping.
 Theoret. Comput. Sci., vol. 7, 1978, 1-24.
- [16] De Groote, H.F.
On varieties of optimal algorithms for the complexity of bilinear mappings. II. Optimal algorithms for 2×2 -matrix multiplication.
 Theoret. Comput. Sci., vol. 7, 1978, 127-148.

- [17] De Groote, H.F.
On varieties of optimal algorithms for the complexity of bilinear mappings. III. Optimal algorithms for the computation of xy and yx where $x, y \in M_2(K)$.
 Theoret. Comput. Sci., vol. 7, 1978, 239-249.
- [18] De Groote, H.F.
Characterization of division algebras of minimal rank and the structure of their algorithm varieties.
 SIAM J. Comput., vol. 12, 1983, 101-117.
- [19] De Groote, H.F. and Heintz, J.
Commutative algebras of minimal rank.
 Linear Algebra and its Appl., vol. 55, 1983, 37-68.
- [20] De Groote, H.F.
Lectures on the complexity of bilinear problems.
 Lectures notes in Computer Science, vol. 245, 1985.
- [21] De Groote, H.F. and Heintz, J.
A lower bound for the bilinear complexity of some semisimple Lie algebras.
 Lectures notes in Computer Science, vol. 229, 1986, 211-222.
- [22] Dobkin, D.
On the arithmetic complexity of a class of arithmetic computations.
 Tech. Rep. 73-23, Dept. of Computer Science, Yale University, 1973.
- [23] Fellmann, A.
Optimal algorithms for finite dimensional simply generated algebras.
 Lectures notes in Computer Science, vol. 229, 1986, 288-295.
- [24] Fiduccia, C.M. and Zalcstein, I.
Algebras having linear multiplicative complexity.
 J. ACM, vol. 24, 1977, 311-331.
- [25] Fischer, P.
Further schemes for combining matrix algorithms.
 Lectures notes in Computer Science, vol. 14, 1974, 428-436.

- [26] Fulton, W..
Curvas Algebraicas.
Editorial Reverte, España , 1971.
- [27] Greub, W..
Multilinear Algebra.
Springer-Verlag, New York, 1967.
- [28] Heintz, J. and Morgerstern, J.
On associative algebras of minimal rank.
Lectures notes in Computer Science, vol. 228, 1986, 1-24.
- [29] Howell, T.D. and Lafon, J.C.
The complexity of the quaternion product.
Cornell University, TR 75-245, 1975.
- [30] Huffman, D.A.
A method for the construction of minimum-redundancy codes.
Proc. IRE, vol. 40, 1952, 1098-1101.
- [31] Kaltofen, E. and Yagati, L.
Improved sparse multivariate polynomial interpolation algorithms.
Lectures notes in Computer Science, vol. 358, 1988, 467-474.
- [32] Kaltofen, E. and Trager, B.
Computing with polynomial given by black boxes for their evaluations: greatest common divisors, factorization, separation of numerators and denominators.
J. Symbolic Computation, vol. 9, 1990, 301-320.
- [33] Karatsuba, A. and Ofman, Y.
Multiplication of multidigit numbers on automata.
Dokl. Acad. Nauk SSSR, vol. 145, 1962, 293-294.
- [34] Keller, W.
Fast algorithms for the characteristic polynomial.
Theoret. Comput. Sci., vol. 36, 1985, 309-317.
- [35] Loos, R.
Computing rational zeros of integral polynomials by p -adic expansion.
SIAM J. Comput., vol. 12, 1983, 286-293.

- [36] Lucchesi, C., Simon I., Simon I., Simon J., and Kowaltowski, T.
Aspectos teóricos da computação.
 Projeto Euclides, 1979.
- [37] Mirwald, R.
The algorithmic structure of $sl(2, k)$.
 Lectures notes in Computer Science, vol. 229, 1986, 274-287.
- [38] Munro, I.
Efficient determination of the transitive closure of a directed graph.
 Information Proc. Letters, vol. 1, 1971, 56-58.
- [39] Ostrowski, A.
On two problems in abstract algebra connected with Horner's rule.
 Studies presented to R. Von Mises, Academic Press, 1954, 40-48.
- [40] Pan, V.Y.
Methods of computing values of polynomials.
 Russian Math. Surveys, vol. 21, 1966, 105-136.
- [41] Pan, V.Y.
Strassen algorithm is not optimal.
 Proc 19th IEEE Symposium on Computer Sci., 1978, 166-176.
- [42] Pan, V.Y.
New fast algorithms for matrix operations.
 SIAM J. Comput., vol. 9, 1980, 321-342.
- [43] Pan, V.Y.
How to multiply matrices faster.
 Lectures notes in Computer Science, vol. 179, 1984.
- [44] Parker, D.
Conditions for optimality of the Huffman algorithm.
 SIAM J. Comput., vol. 9, 1980, 470-489.
- [45] Probert, R.L.
On the complexity of matrix multiplication.
 Tech. Rep. CS 73-27, Computer Science Dept., University of Waterloo,
 Canada, 1973.

- [46] Sieveking, M.
An algorithm for division of powerseries.
Computing, vol. 10, 1972, 153-156.
- [47] Schnorr, C.P.
An extension of Strassen's degree bound.
SIAM J. Comput., vol. 10, 1981, 371-382.
- [48] Schönhage, A. and Strassen, V.
Schnelle multiplikation grosser zahlen.
Computing, vol. 7, 1971, 281-292.
- [49] Schönhage, A.
An elementary proof of Strassen's degree bound.
Theoret. Comput. Sci., vol. 3, 1976, 267-272.
- [50] Schönhage, A.
Partial and total matrix multiplication.
SIAM J. Comput., vol. 10, 1981, 434-455.
- [51] Schönhage, A.
 $\omega < 2.5$ *main ideas behind the new estimates found by D. Coppersmith and S. Winograd.*
Tech. Rep. Universität Tübingen, 1981.
- [52] Strassen, V.
Gaussian elimination is not optimal.
Nuer. Math., vol. 13, 1969, 354-356.
- [53] Strassen, V.
Berechnung und programm I.
Acta Informatica, vol. 1, 1972, 320-335.
- [54] Strassen, V.
Berechnung und programm II.
Acta Informatica, vol. 2, 1973, 64-79.
- [55] Strassen, V.
Vermeidung von divisionen.
J. für die Reine und Angewandte Mathematik, vol. 264, 1973, 184-202.

- [56] Strassen, V.
The asymptotic spectrum of tensors.
 Submitted to J. für die Reine und Angewandte Mathematik.
- [57] Strassen, V.
Relatively bilinear complexity and matrix multiplication.
 Submitted to J. für die Reine und Angewandte Mathematik.
- [58] Terada, R.
Desenvolvimento de algoritmos e complexidade de computação.
 Terceira Escola de Computação, Rio de Janeiro, 1982.
- [59] Toom, A.L.
The complexity of a scheme of functional elements realizing the multiplication of integers.
 Soviet Math. Dokl., vol. 4, 1963, 714-716.
- [60] Valiant, L.G.
General context-free recognition in less than cubic time.
 Tech. Rep. Carnegie-Mellon University, 1974.
- [61] Winograd, S.
On the number of multiplications required to compute certain functions.
 Proc. Nat. Acad. Sci. USA., vol. 58, 1967, 1840-1842.
- [62] Winograd, S.
On the number of multiplications necessary to compute certain functions.
 Comm. Pure Appl. Math., vol. 23, 1970, 165-179.
- [63] Winograd, S.
On multiplication of 2×2 matrices.
 Linear Algebra and its Appl., vol. 4, 1971, 381-388.
- [64] Winograd, S.
Some remarks on fast multiplication of polynomials.
 Traub (editor), Complexity of sequential and parallel numerical algorithms, Academic Press, New York, 1973.
- [65] Winograd, S.
On computing the discrete Fourier transform.
 Math. Comp., vol. 32, 1978, 175-199.

- [66] Winograd, S.
On the multiplicative complexity of the discrete Fourier transform.
Advances in Math., vol. 32, 1979, 83-117.
- [67] Winograd, S.
Arithmetic complexity of computations.
Siam, Philadelphia, Pennsylvania, vol.33, 1980.
- [68] Zippel, R.
Probabilistic algorithm for sparse polynomials.
Lectures notes in Computer Science, vol. 72, 1979.
- [69] Zippel, R.
Interpolation polynomials for their values.
J. Symbolic Computation, vol. 9, 1990, 375-403.