

Algoritmos paralelos eficientes para alguns problemas em processamento de cadeias de caracteres

Siang Wun Song¹

Abstract

In this course we show how efficient parallel algorithms are designed to solve some important string processing problems. One problem considers a sequence or string of characters where a real number is associated to each character and obtains a substring with the maximum sum of the numbers associated to the characters of the substring, as well as an important variation of this problem. Another problem is the obtention of the longest common subsequence of two given sequences and variations of this problem. We first investigate the characteristics of these problems, and then culminate with the presentation of efficient and elegant solutions. It is particularly exciting to learn that the knowledge of these properties enables us to find solutions of more general versions of the basic problem, without increasing the time and space complexity.

Resumo

Mostramos neste curso como algoritmos paralelos eficientes são desenvolvidos para resolver alguns importantes problemas de processamento de seqüências. Um problema considera uma seqüência ou cadeia de caracteres onde um número real está associado a cada caractere e obtém uma subcadeia com soma máxima dos números associados aos caracteres da subcadeia, bem como uma importante variação deste problema. Um outro problema é a obtenção de subseqüências comuns mais compridas de duas seqüências dadas e algumas variações deste problema. Fazemos um estudo preliminar das características destes problemas, culminando com a apresentação de soluções eficientes e elegantes. É particularmente excitante saber que o conhecimento dessas propriedades permitem a obtenção de soluções de versões mais gerais de um problema básico, sem aumentar a complexidade de tempo e espaço.

1.1. Introdução

Este curso tem por objetivo mostrar como são projetados algoritmos paralelos eficientes em tempo e espaço para alguns importantes problemas de processamento de seqüências de caracteres.

¹ Recebe apoio financeiro da FAPESP, Proc. No. 2004/08928-3 e do CNPq, Proc. No. 30.5218/03-4 e 30.5362/06-2

O primeiro problema é denominado *Problema de Todas Subseqüências Maximais*. Este problema tem como entrada uma seqüência ou cadeia de caracteres onde um número real está associado a cada caractere. A saída é uma subcadeia com soma máxima dos números associados aos caracteres da subcadeia, bem como todas as demais subcadeias com somas maximais, a ser definido mais precisamente adiante.

O segundo problema é denominado *Toda-Subcadeia Subseqüências Comuns Mais Longas*, que é uma generalização do problema *Subseqüência Comum Mais Longa*, que obtém a subseqüência mais longa comum a duas seqüências dadas.

Para o projeto de algoritmos paralelos para estes problemas fazemos um estudo preliminar das características destes problemas, culminando com a apresentação de soluções eficientes e elegantes.

Este texto é baseado em vários trabalhos [2, 3, 5, 6, 7] publicados em co-autoria com os Professores Edson Norberto Cáceres e Carlos Eduardo Rodrigues Alves. A eles este autor deve a sua gratidão pela oportunidade de trabalhos conjuntos e por tornar este texto possível.

Na Seção 1.1.1 introduzimos o primeiro problema e na Seção 1.1.2 o segundo problema. O modelo de computação paralela adotado será apresentado na Seção 1.1.3. A seguir, as soluções paralelas dos dois problemas serão apresentadas nas Seções 1.2 e 1.3. Finalmente, concluímos com a Seção 1.4.

1.1.1. Introdução ao problema de todas subseqüências maximais

O primeiro problema considera uma seqüência de caracteres onde a cada caractere está associado um número real. O objetivo deste problema é determinar subcadeias de caracteres em que a soma dos números associados à subcadeia seja maximizada. Como iremos manipular de fato os números associados e não os caracteres propriamente ditos, consideraremos como entrada do problema uma seqüência de números reais, ficando implícito que cada número corresponde a um caractere. Assim, dada uma seqüência de números reais, o *problema de subseqüência máxima* consiste em obter a subseqüência contígua com a máxima soma [13]. Um problema mais geral é o *problema de todas subseqüências maximais* [30], onde queremos achar todas as subseqüências contíguas e disjuntas com soma maximal.

Estes problemas surgem em diversos contextos na Biologia Computacional. Muitas aplicações são apresentadas em [30], por exemplo, para identificar domínios transmembranos em proteínas expressas como uma seqüência de aminoácidos e para descobrir ilhas CpG. Karlin e Brendel [21] definem valores de -5 a 3 para cada um dos 20 aminoácidos. Assim, no contexto do problema, temos uma seqüência de números que variam de -5 a 3. Para a seqüência β_2 -adrenergic receptor humana, subseqüências disjuntas com as maiores somas são obtidas, e estas subseqüências correspondem aos domínios transmembranos do receptor. Csuros [15] menciona ainda outras aplicações que requerem a computação de tais subseqüências, na análise de proteínas e seqüências de

DNA [14], na determinação de *isochores* em seqüências de DNA [19, 23], e na identificação de genes [22].

Algoritmos seqüenciais eficientes de tempo linear são conhecidos para resolver estes problemas [12, 13, 30]. Soluções paralelas são conhecidas apenas para o problema básico, i.e., o problema de subseqüência máxima. Para uma dada seqüência de n números, Wen [35, 26] apresenta um algoritmo EREW PRAM que leva tempo $O(\log n)$ usando $O(n/\log n)$ processadores. Qiu e Akl [28] apresentam um algoritmo paralelo para várias redes de interconexão, tais como o hipercubo, a estrela e a panqueca de tamanho p . Este algoritmo requer tempo $O(n/p + \log p)$ com p processadores. Alves, Cáceres e Song [4] apresentam um algoritmo paralelo BSP/CGM, que requer $O(n/p)$ de tempo de computação e um número constante de rodadas de comunicação.

Na Seção 1.2 apresentaremos um algoritmo paralelo no modelo BSP/CGM que resolve o problema de todas as subseqüências maximais, requerendo um número constante de rodadas de comunicação. Este algoritmo foi apresentado em [6] e num relatório técnico [5]. Dada uma seqüência A de números reais, o algoritmo proposto usa p processadores para achar todas as subseqüências maximais em tempo $O(|A|/p)$, usando espaço $O(|A|/p)$ por processador, e requer um número constante de rodadas de comunicação. Ao contrário da solução paralela para o problema mais básico da obtenção da subseqüência de soma máxima, não é intuitivo que seja possível obter uma solução paralela para o problema mais geral, obtendo todas as subseqüências maximais, em um número constante de rodadas de comunicação, durante as quais no máximo $O(|A|/p)$ de dados são transmitidos entre os processadores.

1.1.2. Introdução ao problema de subseqüência comum mais longa

Introduziremos agora o segundo problema. Dadas duas seqüências ou cadeias de símbolos, a obtenção da subseqüência mais longa comum a ambas as cadeias é um importante problema com aplicações em comparação de seqüências de DNA, compressão de dados, reconhecimento de padrões, etc. [27]. Consideraremos a seguir o problema mais geral denominado *toda-subcadeia subseqüência comum mais longa* e apresentaremos um algoritmo paralelo eficiente em tempo e espaço.

Considere uma seqüência ou cadeia de símbolos de um alfabeto finito. Uma subcadeia de uma cadeia é qualquer fragmento contíguo da cadeia dada. Uma subseqüência de uma cadeia é obtida pela remoção de zero ou mais símbolos da cadeia original. Uma subseqüência pode, portanto, ter símbolos não-contíguos de uma cadeia. Por exemplo, dada a cadeia *lewiscarroll*, um exemplo de uma subcadeia é *scar*, e um exemplo de uma subseqüência é *scroll*. Dadas duas cadeias X e Y , o problema de *subseqüência comum mais longa* (ou *longest common subsequence - LCS*) acha o comprimento da subseqüência mais longa que seja comum a ambas as cadeias. Por exemplo, se $X = \textit{twasbrillig}$ e $Y = \textit{lewiscarroll}$, o comprimento da subseqüência comum mais longa é 5 (e.g. *warll*).

Chamamos a atenção do leitor para a nomenclatura apresentada acima, que difere ligeiramente daquela apresentada para o primeiro problema (Seções 1.1.1 e 1.3). Ao contrário do que foi suposto no problema anterior, uma subsequência no contexto do presente problema não precisa ser contígua.

Dadas as cadeias X e Y de comprimentos m e n , respectivamente, o problema de *toda-subcadeia subsequência comum mais longa* (ou *all-substring longest common subsequence - ALCS*) acha os comprimentos das subsequências comuns mais longas entre X e *qualquer* subcadeia de Y . Vamos apresentar um algoritmo paralelo para ALCS num BSP/CGM (*Coarse-Grained Multicomputer*) com p processadores. Os problemas LCS e ALCS podem ser modelados por meio de um grafo orientado acíclico em grade (*grid directed acyclic graph - GDAG*). O algoritmo proposto acha os comprimentos dos melhores caminhos entre todos os pares de vértices, onde o primeiro vértice do par está na linha superior do GDAG, e o segundo vértice do par está na linha inferior do GDAG. Num BSP/CGM com $p < \sqrt{m}$ processadores, o algoritmo paralelo proposto leva tempo $O(mn/p)$ e espaço por processador de $O(n\sqrt{m})$, com $O(\log p)$ rodadas de comunicação. Segundo nosso conhecimento, este é o melhor algoritmo BSP/CGM para ALCS.

Ao resolver o problema de ALCS, acabamos resolvendo obviamente também o problema LCS, que é menos geral. Entretanto, mesmo considerando o problema mais geral, podemos obter uma complexidade de tempo de $O(mn/p)$, dando origem a um ganho (*speed-up*) linear. Para isso, temos que explorar as propriedades de matrizes totalmente monotônicas, e considerar as similaridades entre linhas das chamadas matrizes D_G , e entre as matrizes consecutivas $MD[i]$. Estas matrizes serão definidas mais tarde. Com essas considerações podemos reduzir a quantidade de informações a ser computadas através da eliminação de redundância. Uma outra preocupação importante é o esforço de usar estruturas de dados compactos para armazenar as informações necessárias, reduzindo assim os tamanhos das mensagens comunicadas entre processadores.

Algoritmos seqüenciais para o problema de LCS foram apresentados em [10, 20, 29]. Algoritmos PRAM foram apresentados para os problemas de LCS e ALCS em [24, 25]. O problema ALCS pode ser resolvido numa PRAM [25] em tempo $O(\log n)$ com $mn/\log n$ processadores, quando $\log^2 m \log \log m \leq \log n$.

Na Seção 1.3 apresentaremos uma solução paralela que é baseada em [3, 7].

1.1.3. O Modelo CGM - Coarse-Grained Multicomputer

O modelo de computação paralela usada neste trabalho é uma versão mais simplificada que o modelo BSP (*Bulk Synchronous Parallel*) [18, 34], denominado modelo CGM (*Coarse-Grained Multicomputer*) [16, 17]. O modelo CGM consta de um conjunto de p processadores cada um com memória local de tamanho $O(n/p)$, onde n denota o tamanho da entrada do problema. Os processadores estão conectados através de uma rede de interconexão arbitrá-

ria. Um algoritmo CGM é constituído de uma alternância entre uma rodada de computação local com uma rodada de comunicação global. Numa rodada de computação local, cada processador processa seus dados na sua memória local, executando um algoritmo seqüencial. Numa rodada de comunicação, cada processador pode enviar $O(n/p)$ de dados e receber $O(n/p)$ de dados.

Um algoritmo eficiente no modelo CGM é aquele que minimiza o número de rodadas de comunicação, bem como o tempo total de computação local. Pode-se mostrar que essa meta leva também a uma melhor portabilidade entre as diversas arquiteturas paralelas [18, 34]. Um bom algoritmo BSP/CGM é aquele que requer um número de rodadas de comunicação independente do tamanho da entrada n . O algoritmo BSP/CGM ideal requer um número constante de rodadas de comunicação. Se isso não for possível, procuramos obter um algoritmo para o qual o número de rodadas de comunicação é independente de n , mas é dependente apenas de p , o número de processadores.

1.2. Problema de todas subseqüências maximais

Apresentaremos inicialmente algumas definições e notações. A seguir, mostraremos os resultados de Ruzzo e Tompa [30] sobre este problema, bem como o algoritmo seqüencial proposto por eles. Para projetar o algoritmo paralelo, iremos primeiro examinar sob que condições subseqüências maximais locais são potenciais candidatos para ser juntadas para formar subseqüências maximais maiores.

1.2.1. Definições preliminares e resultados

Considere uma seqüência A de números reais. Denotamos por A a seqüência e por a_i , $1 \leq i \leq |A|$, seus elementos. Subseqüências de A são indicadas por $A_i^j = (a_{i+1}, \dots, a_j)$. Note que o índice superscrito j indica a posição mais à direita na subseqüência, ao passo que o índice subscripto i é um a menos da posição mais à esquerda. Se o subscripto e o superscrito são iguais, i.e., $i = j$, então a subseqüência é vazia.

Uma subseqüência particular de A pode ser denotada por alguma outra letra maiúscula mas, para evitar confusão, todos os índices referem-se à seqüência A . Para indicar o início e o final de uma seqüência X , subseqüência de A , iremos escrever $L(X)$ e $R(X)$, respectivamente. Para ficar coerente com a notação do último parágrafo, vamos usar $X = A_{L(X)}^{R(X)} = (a_{L(X)+1}, \dots, a_{R(X)})$. Note que $L(X)$ indica uma posição a menos do real início de X .

A concatenação de seqüências X_1, X_2, \dots, X_n será denotada por $\langle X_1, X_2, \dots, X_n \rangle$. Observe que uma seqüência X_i pode consistir de um só número.

A soma dos valores de uma subseqüência X (que iremos também chamar da *score* ou *pontuação* de X) será denotada por $Score(X)$. Se X é vazia, então definimos sua pontuação como sendo zero. A soma de prefixos (*prefix sum* ou *PS*) de A é um conceito importante neste trabalho. Vamos usar $PS(j)$ para denotar a soma dos primeiros j elementos de A , isto é, $PS(j) = Score(A_0^j) = a_1 + a_2 + \dots + a_j$. Consideramos $PS(0) = 0$. Note que $Score(A_i^j) = PS(j) - PS(i)$.

Para uma subsequência $X = A_i^j$, o mínimo e o máximo de todos os valores de $PS(k)$, para $i \leq k \leq j$, será denotado por $Min(X)$ e $Max(X)$, respectivamente.

Neste texto vamos apenas considerar subsequências *contíguas* de uma seqüência principal. Iremos omitir a palavra *contígua* por simplicidade.

1.2.2. Definição do problema

Uma subsequência de X com pontuação máxima, ou simplesmente uma subsequência máxima de X , é aquela que apresenta a maior soma positiva de valores entre todas as subsequências de X . Quando há empate, escolhemos a subsequência de menor comprimento. Se o empate ainda persiste, então a escolha da subsequência particular é irrelevante. Se não há números positivos em X , então consideramos que não há subsequência máxima.

Com essa definição, fica fácil ver que prefixos e sufixos de uma subsequência máxima sempre apresentam somas positivas, uma vez que a remoção de um prefixo ou sufixo com soma não-positiva iria levar a uma subsequência com soma maior.

O problema de achar uma subsequência máxima de A é um problema bem conhecido, e pode ser resolvido por um algoritmo seqüencial em tempo linear [13].

O problema de obter todas as *subseqüências maximais* de A é mais complicado. Em primeiro lugar, precisamos definir o que é uma subsequência maximal. Ruzzo e Tompa [30] definem o conjunto de subsequências maximais de forma procedimental, segundo a seguinte definição recursiva.

Definição 1 Conjunto de subsequências maximais de uma seqüência A .

Dada uma seqüência A de números reais, o conjunto de subsequências maximais de A é vazio se A não tem valores positivos. Caso contrário, seja $\langle A_1, M, A_2 \rangle$ uma decomposição de A em três subsequências, onde M é a subsequência de máxima soma de A (A_1 e A_2 podem ser seqüências vazias). Então, o conjunto de subsequências maximais de A é a união do conjunto $\{M\}$, do conjunto de subsequências maximais de A_1 e o conjunto de subsequências maximais de A_2 .

Para facilitar a compreensão das idéias principais do algoritmo paralelo proposto, vamos adotar a seguinte seqüência como exemplo. Seja a seqüência $A = (a_1, a_2, \dots, a_{22})$ mostrada nas Figuras 1.1 e 1.2. As subsequências maximais de A são $A_0^4 = (5, -3, -1, 5)$, $A_6^9 = (3, 3, 7)$, $A_{10}^{11} = (3)$, e $A_{12}^{19} = (3, -1, 0, 3, -3, 0, 7)$, com as respectivas somas de 6, 13, 3, e 9.

A definição acima leva imediatamente a um algoritmo seqüencial recursivo, que tem no pior caso a complexidade de tempo de $O(n^2)$ para achar todas as subsequências maximais, dada uma seqüência de tamanho n .

Ruzzo e Tompa apresentam as propriedades necessárias e suficientes que uma subsequência X deve possuir para ser maximal na seqüência A . Essas propriedades estão apresentadas no teorema seguinte. Para uma demonstração deste teorema, refira-se a [30].

i	1	2	3	4	5	6	7	8	9	10	11	12	13
a_i	5	-3	-1	5	-9	0	3	3	7	-9	3	-6	3

Figura 1.1. Um exemplo de uma seqüência que será usada neste texto

i	14	15	16	17	18	19	20	21	22
a_i	-1	0	3	-3	0	7	-4	0	-6

Figura 1.2. Continuação da seqüência usada como exemplo

Teorema 1 *Uma subseqüência X é maximal em A se e somente se ela possui as duas propriedades seguintes:*

Propriedade Pr1 *Para cada subseqüência própria Y de X , $Score(Y) < Score(X)$.*

Propriedade Pr2 *Não há superseqüência própria de X que tem Propriedade Pr1.*

Considerando que a seqüência vazia é uma subseqüência de qualquer outra seqüência, segue-se que a soma de uma seqüência com a Propriedade Pr1 deve ser positiva. Subseqüências de A que têm a Propriedade Pr1 serão denominadas *Pr1-subseqüências*.

Podemos reescrever a definição de uma subseqüência maximal em termos dessas propriedades. Usaremos essa nova definição (Definição 2) no decorrer deste texto. A definição anterior (Definição 1) foi apresentada, pois ela é mais natural, ao passo que a nova definição será mais conveniente para entender o algoritmo seqüencial linear de Ruzzo e Tompa, e para entender o algoritmo paralelo que iremos propor.

Definição 2 **Lista de subseqüências maximais de uma seqüência A .** *Dada uma seqüência A de números reais, a lista de subseqüências maximais de A ,*

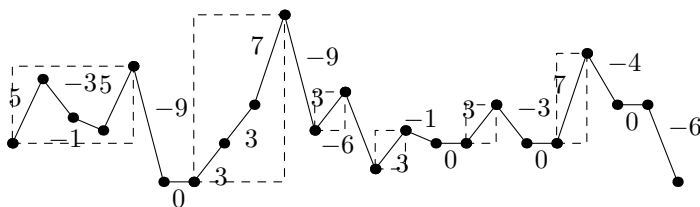


Figura 1.3. Representação gráfica da seqüência $A = (5, -3, -1, \dots, 0, -6)$. Algumas Pr1-subseqüências estão mostradas

denotada por $MList(A)$, é a lista de todas as subseqüências que possuem as Propriedades Pr1 e Pr2, ordenadas com respeito a $L(\cdot)$. Essa lista é indexada a partir de 1 com a subseqüência mais à esquerda.

A Propriedade Pr1 pode também ser apresentada em termos de somas de prefixos.

Lema 1 Uma subseqüência A_i^j é Pr1-subseqüência se e somente se para todo m , $i < m < j$, $PS(i) < PS(m) < PS(j)$.

Prova: Se A_i^j é uma Pr1-subseqüência, $Score(A_i^j) > Score(A_i^m)$. Então, $PS(j) - PS(i) > PS(m) - PS(i)$ e $PS(j) > PS(m)$. Também $Score(A_i^j) > Score(A_m^j)$, o que leva a $PS(i) < PS(m)$.

Se $PS(i) < PS(m) < PS(j)$ para todo m , $i < m < j$, qualquer A_r^l que é uma subseqüência própria de A_i^j tem $Score(A_r^l) = PS(r) - PS(l) < PS(j) - PS(i) = Score(A_i^j)$.

□

Uma representação gráfica é útil aqui. Traçamos a função $PS(\cdot)$, de tal maneira que valores positivos (negativos) na seqüência são representados por linhas de segmentos ascendentes (descendentes). Veja a Figura 1.3 para o exemplo. Uma Pr1-subseqüência X será indicada por uma caixa retangular com $(L(X), PS(L(X)))$ e $(R(X), PS(R(X)))$ com os cantos inferior esquerdo e superior direito, respectivamente. A curva traçada toca a caixa retangular somente nestes cantos. Note que as três primeiras subseqüências na Figura 1.3 são subseqüências maximais de A , mas as três últimas não são. (Elas são subseqüências da mesma A -maximal, a saber, A_{12}^{19}).

Dizemos que A_i^j ($i < j$) é um Pr1-prefixo se $PS(i) < Min(A_{i+1}^j)$, e ele é um Pr1-sufixo se $Max(A_i^{j-1}) < PS(j)$. Uma Pr1-subseqüência é ao mesmo tempo um Pr1-prefixo e um Pr1-sufixo.

Corolário 1 Se P é um Pr1-prefixo e S é um Pr1-sufixo, $\langle P, S \rangle$ é uma Pr1-subseqüência se e somente se $Min(P) < Min(S)$ e $Max(P) < Max(S)$.

1.2.3. Alguns resultados sobre subseqüências maximais

Damos agora alguns resultados que serão úteis na descrição do algoritmo seqüencial e do algoritmo paralelo, a serem apresentados. Primeiro vamos apresentar alguns lemas de [30].

Lema 2 Qualquer Pr1-subseqüência de uma seqüência A está contida (não necessariamente propriamente) em uma subseqüência maximal de A .

Prova: Suponha que a afirmação não seja verdadeira. Seja X a maior Pr1-subseqüência de A não-contida em qualquer subseqüência maximal. Então, X não é maximal, não possui a Propriedade Pr2, e portanto deve estar contida em uma Pr1-subseqüência maior de A , o que leva a uma contradição.

□

Lema 3 *Dada uma seqüência A , quaisquer duas subseqüências maximais distintas de A não se sobrepõem ou tocam uma na outra.*

Prova: Suponha que esta asserção não seja verdadeira. Sejam A_i^k e A_j^l duas subseqüências maximais distintas que violam essa asserção. Uma subseqüência maximal não pode estar propriamente contida na outra (Propriedade Pr2). Assim, sem perda de generalidade, podemos considerar $i < j \leq k < l$. Pelo Lema 1 aplicado a A_i^k temos $PS(i) < PS(j)$, e o mesmo lema aplicado a ambas as subseqüências mostra que $PS(i) < PS(m)$ para todo $m, i < m < l$. Analogamente, podemos provar que $PS(m) < PS(l)$ para todo $m, i < m < l$. Aplicando o Lema 1 na outra direção, concluímos que A_i^k é uma Pr1-subseqüência, portanto A_i^k e A_j^l não possuem a Propriedade Pr2, uma contradição.

□

Tanto o algoritmo seqüencial como o algoritmo paralelo são baseados em achar listas de subseqüências maximais em segmentos da seqüência original A . Considere uma subseqüência X de A . Diremos que uma subseqüência é uma *subseqüência X -maximal*, ou simplesmente uma *X -maximal*, se ela é maximal em X , isto é, ela é uma Pr1-subseqüência e não tem superseqüência própria que seja uma Pr1-subseqüência de X . Queremos achar o conjunto de todas A -maximais.

Baseado no lema anterior, diremos que uma A -maximal está à esquerda de outra se seu $L(\cdot)$ é menor.

Vamos aplicar os lemas anteriores a qualquer subseqüência de A , não somente a A propriamente.

Lema 4 *Seja $Z = \langle X, Y \rangle$ para algumas X e Y não-vazias. Então, há no máximo uma Z -maximal M que sobrepõe ambas X e Y . Se existe tal M , então ela tem uma X -maximal como prefixo e uma Y -maximal como sufixo. As X -maximais à esquerda de M e as Y -maximais à direita de M são também Z -maximais.*

Prova: Aplicando o Lema 3 a Z , fica óbvio que não mais que uma Z -maximal sobrepõe X e Y . Suponhamos que existe uma tal Z -maximal M . $X = A_i^j$, $Y = A_j^k$ e $M = A_{m_1}^{m_2}$ para algum $0 \leq i \leq m_1 < j < m_2 \leq k \leq |A|$. Provamos agora as afirmações referentes a X . As afirmações sobre Y são provadas analogamente.

$PS(m_1)$ é a soma de prefixos mínima em M . Se n é o menor valor em $]m_1, j]$, tal que $PS(n)$ é máximo, então $A_{m_1}^n$ é uma Pr1-subseqüência de X . Pelo Lema 2, $A_{m_1}^n$ deve estar contida em uma X -maximal. Esta X -maximal é uma Pr1-subseqüência de Z , portanto, ela deve estar contida em uma Z -maximal, que só pode ser M . Por esta razão e pela escolha de n , vemos que $A_{m_1}^n$ é uma X -maximal que é um prefixo de M .

Qualquer X -maximal que está à esquerda de M é uma Pr1-subseqüência. Se ela não é uma Z -maximal, então existe uma Pr1-subseqüência de Z que

a contém, e a maximalidade em X proíbe esta Pr1-subseqüência maior estar contida em X . Portanto, esta Pr1-subseqüência sobrepõe X e Y , contradizendo a unicidade de M .

□

O Lema 4 é importante para o algoritmo seqüencial, pois ele mostra que é possível construir $MList(A)$ de forma incremental. Tendo um prefixo X de A e suas subseqüências maximais, podemos estender este prefixo para a direita, preservando algumas das X -maximais, e eventualmente criando uma outra subseqüência maximal que envolve a extensão e as X -maximais mais à direita. O algoritmo seqüencial concatena simplesmente um outro número a X em cada passo, portanto $|Y| = 1$. Mostraremos estes detalhes em breve.

O Lema 4 é também importante para o algoritmo paralelo a ser apresentado. A seqüência A é dividida em subseqüências que são tratadas separadamente. Suas subseqüências maximais são usadas mais tarde para obter as A -maximais.

O algoritmo paralelo trata do seguinte subproblema: dada uma subseqüência X de A e sua lista de subseqüências maximais $MList(X)$, achar, se possível, uma X -maximal que é um prefixo (ou sufixo) de uma A -maximal maior. Isso claramente envolve $MList(X)$ e o restante da A . Entretanto, algumas X -maximais não precisam ser consideradas como possíveis prefixos ou sufixos de A -maximais maiores, não importando o que está fora de X . A eficiência do nosso algoritmo paralelo está baseada nessa importante noção. Portanto, vamos formalizá-la nas definições e lemas seguintes. Iremos primeiro tratar de candidatos a prefixos.

Definição 3 ($PList(X)$) *Dada uma subseqüência X de A , $PList(X)$ é a lista ordenada de todas as X -maximais, com a exceção daquelas X -maximais M para as quais uma das duas condições seguintes está satisfeita.*

1. $Min(M) \geq PS(R(X))$ ou
2. existe uma X -maximal N à direita de M tal que $Min(M) \geq Min(N)$.

Os elementos de $PList(X)$ estão indexados começando com 1 com a subseqüência mais à esquerda.

Informalmente, $PList(X)$ nos dá a lista de todas as X -maximais que são potenciais candidatos para ser juntadas à direita para dar origem a maximais maiores. Note que excluímos de $PList(X)$ aquelas X -maximais (satisfazendo às condições 1 e 2) que nunca podem fornecer maximais maiores. Considere $X = A_0^{14}$ da seqüência usada como exemplo (ver Figuras 1.1 e 1.2 e Figura 1.3). Há quatro X -maximais, a saber, A_0^4 , A_6^9 , A_{10}^{11} , e A_{12}^{13} (indicadas pelas primeiras quatro caixas retangulares da Figura 1.3). A_0^4 não pertence a $PList(X)$ por causa da condição 1. A_{10}^{11} não pertence a $PList(X)$ por causa das condições 1 e 2. Portanto, $PList(X) = (A_6^9, A_{12}^{13})$.

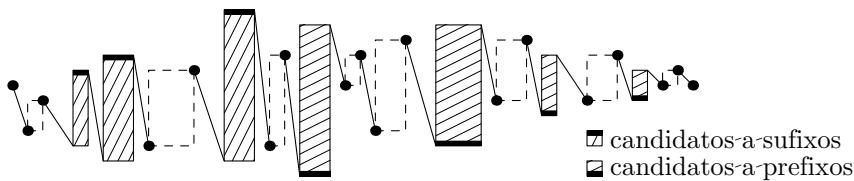


Figura 1.4. Representação gráfica de uma seqüência X , $MList(X)$, $PList(X)$ e $SList(X)$. A primeira (última) maximal não é um candidato a sufixo (prefixo) por causa da primeira condição da definição. As outras maximais que não são candidatas recaem na segunda condição. As linhas descendentes representam seqüências de números não-positivos.

Lema 5 Se X é uma subseqüência de A , $PList(X)$ contém todas as X -maximais que podem ser um prefixo próprio de uma A -maximal.

Prova: Para uma X -maximal M , qualquer uma das duas condições na Definição 3 implica a existência de $i \in]L(M), R(X)]$, tal que $PS(L(M)) \geq PS(i)$. Assim sendo, nenhuma A -maximal pode estender de M passando $R(X)$, porque ela iria violar a Propriedade Pr1. Portanto, as X -maximais removidas de $PList(X)$ não são prefixos próprios de qualquer A -maximal.

□

Lema 6 Se M é uma seqüência em $PList(X)$ e $i \in]L(M), R(X)]$, então $Min(M) < PS(i)$, isto é, $A_{L(M)}^{R(X)}$ é um Pr1-prefixo.

Prova: Suponha que seja possível achar um i no intervalo especificado tal que $PS(L(M)) \geq PS(i)$. Pegue o maior i possível. A Condição 1 da Definição 3 não iria permitir M em $PList(X)$ se $i = R(X)$, assim $i < R(X)$. A escolha de i garante que $PS(i+1) > PS(i)$, assim A_i^{i+1} é uma Pr1-seqüência, e ela deve estar contida em alguma X -maximal N (conforme o Lema 2). N tem que estar à direita de M e $Min(M) \geq PS(i) \geq Min(N)$, mas então a Condição 2 da Definição 3 não permitiria M em $PList(X)$, uma contradição.

□

Lema 7 Se M é uma seqüência em $PList(X)$ e $i \in]R(M), R(X)]$, então $Max(M) \geq PS(i)$.

Prova: Suponha que seja possível achar um i no intervalo especificado, tal que $Max(M) < PS(i)$. Pegue o menor i possível. Pegue o maior valor de j , tal que $L(M) \leq j < i$ e $PS(j)$ seja mínimo no intervalo. Fica claro pelo Lema 1 que A_j^i tem a Propriedade Pr1. Portanto, há uma X -maximal N que a contém. Como X -maximais distintas não podem sobrepor-se (Lema 3), N deve estar à direita de M . Pela escolha de j devemos ter $Min(N) \leq Min(M)$, mas então M não deve estar em $PList(X)$ pela condição 2 da Definição 3, uma contradição.

□

Uma consequência direta dos lemas anteriores é que $PList(X)$ está em ordem não-crescente de $Max(\cdot)$ e em ordem estritamente crescente de $Min(\cdot)$. A Figura 1.4 ilustra $PList(X)$ (e $SList(X)$, a ser definido em breve).

Para o algoritmo paralelo precisamos de uma definição similar para possíveis sufixos de A -maximais. A definição e os lemas correspondentes são agora dadas, sem provas, que são semelhantes às anteriores. Note o papel intercambiável de $Max(\cdot)$ e $Min(\cdot)$, “esquerda” e “direita” etc.

Definição 4 ($SList(X)$) Dada uma subseqüência X de A , $SList(X)$ é uma lista ordenada de todas as X -maximais, com a exceção daquelas X -maximais N para as quais uma das duas condições seguintes está satisfeita.

1. $Max(N) \leq PS(L(X))$ ou
2. existe uma X -maximal M à esquerda de N tal que $Max(N) \leq Max(M)$.

Os elementos de $SList(X)$ estão indexados começando com 1 com a subseqüência mais à direita.

Lema 8 Se X é uma subseqüência de A , $SList(X)$ contém todas as X -maximais que podem ser um sufixo próprio de uma A -maximal.

Lema 9 Se N é uma seqüência em $SList(X)$ e $i \in [L(X), R(N)[$, então $PS(i) < Max(N)$, isto é, $A_{L(X)}^{R(N)}$ é um Pr1-sufixo.

Lema 10 Se N é uma seqüência em $SList(X)$ e $i \in [L(X), L(N)[$, então $PS(i) \geq Min(N)$.

Note que no máximo uma X -maximal pode pertencer simultaneamente a $PList(X)$ e $SList(X)$, a saber, a subseqüência máxima de X . Qualquer outro elemento de $SList(X)$ deve estar à esquerda de qualquer outro elemento de $PList(X)$. Ver a Figura 1.4 para uma ilustração de $PList(X)$ e $SList(X)$ (onde estas listas são disjuntas).

1.2.4. O algoritmo seqüencial

Apresentamos agora o Algoritmo 1, uma versão modificada do algoritmo seqüencial de Ruzzo e Tompa [30]. Há várias diferenças entre o Algoritmo 1 e o original de [30]. O procedimento é apresentado de forma mais explícita aqui, fazendo uso de *arrays* para facilitar a análise e usando um nível de aninhamento de laços a menos, mas as idéias principais e o desempenho permanecem os mesmos. Também deixamos explícita a construção de *PList*(.), o qual é implicitamente usado no algoritmo original de Ruzzo e Tompa como uma lista ligada auxiliar.

Algorithm 1 Subseqüências Maximais (Algoritmo Seqüencial)

Require: Seqüência $A = (a_1, a_2, \dots, a_{|A|})$

Ensure: Arrays *Ml* e *Pl*, com n_m e n_p elementos, respectivamente. *s* guarda a soma de prefixos.

```

1:  $n_m \leftarrow 0, n_p \leftarrow 0, s \leftarrow 0$ 
2: for  $i \leftarrow 1$  a  $|A|$  do
3:    $s \leftarrow s + a_i$ 
4:   if  $a_i < 0$  then
5:     while  $n_p > 0$  and  $Min(Ml[Pl[n_p]]) \geq s$  do
6:        $n_p \leftarrow n_p - 1$  {Pop candidatos a prefixos}
7:     end while
8:   end if
9:   if  $a_i > 0$  then
10:    {Push nova seqüência formada por  $a_i$  somente (dado parcial, podendo
    ser descartado)}
11:     $n_m \leftarrow n_m + 1$ 
12:     $Min(Ml[n_m]) \leftarrow s - a_i$  {s anterior}
13:     $L(Ml[n_m]) \leftarrow i - 1$ 
14:     $Pl[n_p + 1] \leftarrow n_m$ 
15:    while  $n_p > 0$  e  $Max(Ml[Pl[n_p]]) < s$  do
16:       $n_p \leftarrow n_p - 1$  {Pop candidatos a prefixos, procurando pelo melhor para
      juntar com  $a_i$ }
17:    end while
18:     $n_p \leftarrow n_p + 1$ 
19:    { $Pl[n_p]$  é o melhor candidato a prefixo}
20:     $n_m \leftarrow Pl[n_p]$  {Pop seqüências}
21:    {Complete o dado da seqüência do topo}
22:     $R(Ml[n_m]) \leftarrow i$ 
23:     $Max(Ml[n_m]) \leftarrow s$ 
24:  end if
25: end for

```

A entrada do algoritmo é a seqüência *A* e a saída é *MList*(*A*) (ou simplesmente indicada como *Ml* no algoritmo) e *PList*(*A*) (ou simplesmente *Pl*). Am-

bas as listas são implementadas como *arrays* com o primeiro índice 1 e usadas como *pilhas* com índice 1 referindo-se ao fundo da pilha. *Pl* irá armazenar os índices dos elementos em *Ml* enquanto que *Ml* irá armazenar os dados sobre as *A*-maximais ($L(\cdot)$, $R(\cdot)$, $Max(\cdot)$ e $Min(\cdot)$).

Teorema 2 *Dada uma seqüência de números A , Algoritmo 1 computa $MList(A)$ e $PList(A)$ corretamente usando $O(|A|)$ tempo e espaço.*

Prova: Vamos provar que ao final de cada iteração do laço na linha 2 *Ml* e *Pl* representam $MList(A_0^i)$ e $PList(A_0^i)$, respectivamente. Note que no começo da primeira iteração temos $i = 1$ e A_0^0 é a subseqüência vazia. Tanto *Ml* como *Pl* são vazias, representando $MList(A_0^0)$ e $PList(A_0^0)$.

Considere $X = A_0^{i-1}$ e $Z = A_0^i$. Mostramos agora que o corpo do laço constrói $MList(Z)$ e $PList(Z)$ baseado em $MList(X)$ e $PList(X)$. Após a linha 3, $s = PS(R(Z))$ e $s - a_i = PS(R(X))$.

Usando o Lema 4, procuramos por uma *Z*-maximal única que sobrepõe *X* e termina em a_i . Se a_i é não-positivo, então pelo Lema 1 ele não pode ser o sufixo de qualquer maximal, assim $MList(Z) = MList(X)$. Baseado nisso, $PList(Z)$ deve ser o mesmo que $PList(X)$, exceto para as *X*-maximais que devem ser removidas de acordo com a primeira condição da Definição 3. Portanto, o laço na linha 5 remove todas as *Z*-maximais que têm $Min(\cdot)$ não menor que $PS(R(Z))$.

Se $a_i > 0$, então ele deve ser incluído em alguma *Z*-maximal. Linhas 11 até 14 introduzem uma nova seqüência, contendo somente a_i , em *Ml* e *Pl*. Esta seqüência não é necessariamente uma *Z*-maximal. Somente os dados que se referem ao início desta seqüência ($L(\cdot)$ e $Min(\cdot)$) são introduzidos em *Ml*. Agora o algoritmo procura pela maior Pr1-subseqüência possível de *Z* que contém a_i , isto é, uma *Z*-maximal.

Aplicando os Lemas 4 e 5 a *Z*, os possíveis prefixos para esta *Z*-maximal são os elementos de $PList(X)$ (ou a_i propriamente). Pelo Lema 6, se *M* é uma seqüência em $PList(X)$, então $A_{L(M)}^{i-1}$ é um Pr1-prefixo. A seqüência formada por a_i sozinho é um Pr1-sufixo. Assim sendo, podemos aplicar o Corolário 1: $A_{L(M)}^i$ é uma Pr1-subseqüência se e somente se $Min(M) < PS(i-1)$ e $Max(M) < PS(i)$. A primeira desigualdade é válida pela Definição 3 (ver a Condição 1). A segunda desigualdade requer uma busca em $PList(X)$.

Pelo Lema 7, se um elemento *M* de $PList(X)$ satisfaz à segunda desigualdade, todos os elementos à direita de *M* também satisfazem. Estamos interessados no elemento mais à esquerda de $PList(X)$ que satisfaz à desigualdade, pois ele leva à maior Pr1-seqüência possível. O laço na linha 15 faz uma busca por esta seqüência. Uma vez encontrada, o dado relativo ao seu final ($R(\cdot)$ e $Max(\cdot)$) é mudado para refletir a extensão da seqüência até a_i (linhas 22 e 23). Todas as seqüências em $MList(X)$ de *M* ao final de $MList(X)$ são descartadas e substituídas pela nova seqüência (linha 20) e as seqüências à esquerda de *M* são mantidas, de acordo com o Lema 4.

Finalmente, note que o algoritmo remove de Pl somente as seqüências que eram absorvidas pela nova seqüência, que ainda está no *array*. Pela Definição 3, não há razão em remover qualquer das outras seqüências, pois nenhuma nova seqüência com o menor $Min(\cdot)$ foi introduzida e $PS(Z)$ é maior que $PS(X)$ ($a_i > 0$), assim terminamos com $Pl = PList(Z)$.

Note que o laço na linha 15 pode falhar no primeiro teste, indicando que nenhum elemento de $PList(X)$ pode ser um prefixo próprio de uma Z -maximal. Neste caso, a seqüência introduzida nas linhas 11 até 14 é usada como M no parágrafo anterior. $PList(Z)$ é igual a $PList(X)$, com a inclusão desta última seqüência. Nenhuma outra seqüência necessita ser eliminada.

Vamos provar que o algoritmo usa somente $O(|A|)$ tempo e espaço. Ml e Pl têm aproximadamente $|A|/2$ elementos no pior caso. Fica claro então a linearidade no espaço. O laço principal na linha 2 executa $|A|$ iterações. Cada comando neste laço executa claramente em tempo constante, exceto os laços nas linhas 5 e 15. Usando análise amortizada, observamos que n_p nunca fica negativo. É fácil observar que o número total de iterações executadas por estes dois laços é limitado pelo número de vezes n_p é incrementado na linha 18, que é $O(|A|)$. Concluimos que o algoritmo executa em $O(|A|)$ tempo e espaço. □

1.2.5. O algoritmo paralelo

Apresentamos nas próximas seções o algoritmo BSP/CGM para achar todas as subseqüências maximais de uma seqüência A usando p processadores, denominados de P_i , $i \in [1, p]$. Supomos que A está dividida em p subseqüências, cada uma de tamanho $l = \lceil |A|/p \rceil$, exceto a última, que pode ser menor. Denotamos estas subseqüências por $AP_i = A_{l(i-1)}^{l_i}$.

No início do procedimento, para todo $i \in [1, p]$ AP_i já está armazenado na memória local do processador P_i . No final do procedimento, o processador P_i conterà a informação (posição e pontuação) de todas as A -maximais que começam ou terminam dentro de AP_i .

O algoritmo paralelo proposto obtém todas as subseqüências maximais de uma seqüência A distribuída nas p memórias locais) em tempo $O(|A|/p)$, usando $O(|A|/p)$ espaço local e $O(1)$ rodada de comunicação.

1.2.6. Obtenção das maximais locais

Os resultados da Seção 1.2.4 são resumidos no seguinte lema.

Lema 11 *Em $O(|A|/p)$ tempo e espaço e usando uma rodada de comunicação de tamanho $O(p)$ (ou seja, $O(p)$ dados são transmitidos), cada processador P_i ($i \in [1, p]$) pode obter a seguinte informação:*

- suas listas locais de maximais ($MList(AP_i)$), candidatas a prefixos ($PList(AP_i)$) e candidatas a sufixos ($SList(AP_i)$).
- $PS(L(AP_j))$, $Min(AP_j)$ e $Max(AP_j)$ para todo $j \in [1, p]$.

Prova: Quando executado por processador P_i , Algoritmo 1 fornece $MList(AP_i)$, $PList(AP_i)$ e $Score(AP_i)$, mas, sem a informação dos outros processadores, ele tem que supor que $PS(L(AP_i)) = 0$. O valor real não é importante para a construção das listas, mas ele deverá ser adicionado aos valores das somas de prefixos nestas listas.

Usando a Definição 4, uma simples varredura em $MList(AP_i)$ fornece $SList(AP_i)$. Esta varredura permite a obtenção de $Min(AP_i) - PS(L(AP_i))$ e $Max(AP_i) - PS(L(AP_i))$.

Os dois últimos valores e $Score(AP_i)$ podem ser enviados a todos os processadores em uma rodada de comunicação de tamanho $O(p)$. Todos os processadores terão $Score(AP_j)$ para todo $j \in [1, p]$ e poderão calcular $PS(L(AP_j))$, $Min(AP_j)$ e $Max(AP_j)$ para todo $j \in [1, p]$. Isso pode parecer ineficiente, mas sob nossas considerações, é melhor do que paralelizando esta operação simples e gastando mais tempo em comunicação.

Cada processador pode então atualizar os valores das somas de prefixos nas suas três listas de resultados. É fácil ver que todas as operações descritas aqui podem ser feitas em $O(|A|/p)$ tempo e espaço.

□

1.2.7. Procedimento básico para juntar listas de maximais

Mostramos agora como $MList(Z)$ pode ser obtida de $MList(X)$, $MList(Y)$, $PList(X)$ e $SList(Y)$ quando $Z = \langle X, Y \rangle$. O procedimento mostrado aqui é a base do nosso algoritmo paralelo, mas o leitor deve ser avisado de que o algoritmo não está baseado nos passos sucessivos de junção dois-a-dois de subseqüências. Uma tal estratégia iria dar origem a $O(\log p)$ rodadas de comunicação e resultaria em um ganho (*speed-up*) sublinear. Mostraremos mais tarde como os dados parciais descritos na Seção 1.2.6 são usados numa operação de junção global, com um número constante de rodadas de comunicação.

O lema seguinte apresenta a condição para duas subseqüências maximais locais serem juntadas para formar uma maior.

Lema 12 Dados $M \in PList(X)$ e $N \in SList(Y)$, $A_{L(M)}^{R(N)}$ é uma Pr1-subseqüência se e somente se $Min(M) < Min(N)$ e $Max(M) < Max(N)$.

Prova: Seja $m = L(M)$, $l = R(X) = L(Y)$, $n = R(N)$. Os Lemas 6 e 9 estabelecem que A_m^l e A_l^n são respectivamente um Pr1-prefixo e um Pr1-sufixo. Os Lemas 7 e 10, juntamente com o Lema 1, aplicados a M e N , estabelecem que $Max(M) = Max(A_m^l)$ e $Min(N) = Min(A_l^n)$. O lema segue do Corolário 1 aplicado a $A_m^n = \langle A_m^l, A_l^n \rangle$.

□

Os Lemas 5 e 8 mostram que podemos procurar por uma Z -maximal que sobrepe X e Y usando somente $PList(X)$ e $SList(Y)$. Algoritmo 2 faz isso.

Usamos $Pl = PList(X)$ e $Sl = SList(Y)$, indexando-os como indicados nas Definições 3 e 4. O algoritmo retorna os índices dos candidatos a prefixos e sufixos escolhidos da nova Z -maximal. Neste algoritmo usamos os elementos de Pl e Sl das seqüências reais, não como índices a listas de maximais, por simplicidade.

Algorithm 2 Junção de duas Listas de Maximais

Require: Listas Pl e Sl , com $|Pl|$ e $|Sl|$ candidatos, respectivamente.
Ensure: *Flag* f que indica se uma nova maximal foi achada, com os índices i_p e i_s dos candidatos que definem esta maximal.

```

1:  $i_p \leftarrow 1, i_s \leftarrow 1, f \leftarrow false$ 
2: while  $i_p \leq |Pl|$  and  $i_s \leq |Sl|$  and not  $f$  do
3:   if  $Max(Pl[i_p]) \geq Max(Sl[i_s])$  then
4:      $i_p \leftarrow i_p + 1$ 
5:   else if  $Min(Pl[i_p]) \geq Min(Sl[i_s])$  then
6:      $i_s \leftarrow i_s + 1$ 
7:   else
8:      $f \leftarrow true$ 
9:   end if
10: end while

```

Lema 13 Dados $Z = \langle X, Y \rangle$, $Pl = PList(X)$ e $Sl = SList(Y)$, Algoritmo 2 obtém a única Z -maximal que sobrepõe X e Y , se ela existir, em $O(|Pl| + |Sl|)$ tempo e $O(1)$ espaço adicional.

Prova: As complexidades de tempo e espaço do Algoritmo 2 são claramente as indicadas. Precisamos entretanto provar que ele realmente acha a Z -maximal, se ela existir. Note que, pelo Lema 4, esta Z -maximal é única.

Provamos por indução a seguinte afirmação: no momento em que o teste do laço é executado, nenhuma Z -maximal existe com prefixo $Pl[i]$ com $i \in [1, i_p[$ ou com sufixo $Sl[j]$ com $j \in [1, i_s[$.

A afirmação é claramente verdadeira para o primeiro teste, uma vez que não há candidatos a prefixo ou sufixo no intervalo especificado. Suponha a afirmação verdadeira para uma particular iteração. O comando condicional dentro do laço será executado como se segue: se o primeiro teste resultar em *true*, então não há candidatos a sufixos restantes em Sl com $Max(\cdot)$ maior que $Max(Pl[i_p])$ (Lema 9). Pelos Lemas 8 e 12 e a hipótese de indução, concluímos que $Pl[i_p]$ não é um prefixo próprio de qualquer Pr1-subseqüência válida de Z , assim i_p é aumentado e a afirmação permanece verdadeira. A análise para o caso em que o segundo teste resultar em *true* é similar.

Se o laço termina com $f = false$, então não há nova Z -maximal. Se o laço termina com $f = true$, então $Pl[i_p]$ e $Sl[i_s]$ satisfazem às condições do Lema 12 e, portanto, definem uma Pr1-subseqüência de Z . A afirmação que acaba de

ser demonstrada mostra que não há outra Pr1-subseqüência que pode conter propriamente aquela definida por $PI[i_p]$ e $SI[i_s]$. Assim, esta subseqüência tem a Propriedade Pr2 e é uma Z-maximal.

□

1.2.8. Rotulando (Tagging) os candidatos locais

O algoritmo paralelo executa um passo de junção ou união, usando um número constante de rodadas de comunicação, envolvendo todas as maximais locais achadas por cada processador no passo local. O passo de junção é baseado na simples observação de que uma maximal não-local deve começar dentro de algum AP_i e terminar em algum AP_j com $1 \leq i < j \leq p$. Portanto, ela deve ter alguma seqüência em $PList(AP_i)$ como prefixo e alguma seqüência em $SList(AP_j)$ como sufixo.

O problema é achar um conjunto *relevante* de Pr1-subseqüências de A que cruzam fronteiras de processadores. Por *relevante* queremos dizer que todas as A -maximais que cruzam fronteiras de processadores devem estar contidas neste conjunto. Num último passo teremos que somente escolher as Pr1-subseqüências que não estão contidas numa outra.

Dizemos que um candidato a prefixo e um candidato a sufixo *casam* ou *combinam* (*match*) se eles definem uma Pr1-subseqüência de A . A definição seguinte estabelece as condições de um casamento.

Lema 14 Para $M \in PList(AP_i)$ e $N \in SList(AP_j)$, $1 \leq i < j \leq p$, $A_{L(M)}^{R(N)}$ (a seqüência que tem M como prefixo, N como sufixo e contém AP_k , $i < k < j$) é uma Pr1-subseqüência se e somente se $Min(M) < Min(N)$, $Max(M) < Max(N)$, $Min(M) < \min_{i < k < j} Min(AP_k)$ e $Max(N) > \max_{i < k < j} Max(AP_k)$.

Prova: A prova é análoga à do Lema 12. As condições extras envolvendo AP_k , $i < k < j$, estão relacionadas ao Lema 1.

□

Depois do passo local descrito na Seção 1.2.6 os processadores não podem determinar quais candidatos combinam, pois eles somente têm acesso a suas próprias listas de candidatos. Entretanto, dado um particular candidato a prefixo ou sufixo, as condições extras do Lema 14 permitem a determinação dos processadores onde um casamento para este candidato pode ser achado. Assim, o primeiro passo na operação de junção global é rotular (*tag*) cada candidato com o(s) número(s) do(s) processador(es) que pode(m) conter um tal casamento para ele. Veremos que cada candidato recebe no máximo um rótulo, com algumas poucas exceções a serem vistas.

Veremos como rotular o candidato a prefixo de AP_i . O caso para candidatos a sufixos é análogo. Para simplificar a discussão, vamos considerar uma lista de possíveis rótulos para estes prefixos, denominados $PTagList(i)$. Esta

lista é construída como se segue. Inicialmente, $P\text{TagList}(i)$ contém um elemento correspondendo a cada processador com número maior que i . Para o processador $j \in]i, p]$ um elemento T conterá a seguinte informação: o número do processador representado por $\text{tag}(T) = j$ (o próprio rótulo), a máxima e a mínima das somas de prefixos dentro da subsequência correspondente de A , $\text{Max}(T) = \text{Max}(AP_j)$ e $\text{Min}(T) = \text{Min}(AP_j)$. T conterá também a mínima das somas de prefixos em todas as seqüências locais de AP_{i+1} a AP_{j-1} : $\text{Min}^*(T) = \min_{i < k < j} \text{Min}(AP_k)$ (∞ if $j = i + 1$). $\text{Min}^*(T)$ será útil na busca por rótulos de acordo com o Lema 14. A lista inteira de rótulos é facilmente construída em tempo $O(p)$.

Desta lista eliminamos os elementos que têm um rótulo tal que existe j , $i < j < k$ e $\text{Max}(AP_j) \geq \text{Max}(AP_k)$. Isso se deve ao fato de que qualquer candidato a sufixo $N \in S\text{List}(AP_k)$ teria $\text{Max}(N) \leq \text{Max}(AP_k) \leq \text{Max}(AP_j)$, não sendo combinável com qualquer candidato a prefixo de AP_i (Lema 14). A eliminação de todas as seqüências nesta condição leva tempo $O(p)$.

A lista final é indexada em ordem decrescente de acordo com tag , começando com o índice 1. Esta indexação é usada para tornar $P\text{TagList}(i)$ semelhante a uma lista de candidatos a sufixos. O algoritmo de rotulamento (*tagging*), a ser apresentado em breve, é similar ao Algoritmo 2.

Baseado no que foi apresentado, afirmamos o seguinte.

Afirmção 1 Para qualquer $i \in [1, p]$, $P\text{TagList}(i)$ contém todos os rótulos que representam seqüências locais que podem ter um candidato a sufixo que casa com algum candidato a prefixo de AP_i . A indexação de $P\text{TagList}(i)$ é em ordem decrescente de $\text{tag}(\cdot)$, uma ordem (estritamente) decrescente de $\text{Max}(\cdot)$ e uma ordem não-decrescente de $\text{Min}^*(\cdot)$.

Vamos considerar o rotulamento de um particular candidato a prefixo M .

Observação 1 Quando comparando M com algum $T \in P\text{TagList}(i)$, os casos seguintes podem ocorrer:

Caso 1 $\text{Max}(M) \geq \text{Max}(T)$. Isso desqualifica $\text{tag}(T)$ para M (Lema 14). Como $P\text{TagList}(i)$ é ordenado em ordem decrescente de $\text{Max}(\cdot)$, os seguintes rótulos em $P\text{TagList}(i)$ são também desqualificados.

Caso 2 $\text{Max}(M) < \text{Max}(T)$. Há dois casos a considerar:

Case 2.i $\text{Min}(M) \geq \text{Min}^*(T)$. Isso desqualifica $\text{tag}(T)$ para M (de novo pelo Lema 14). Como $P\text{List}(AP_i)$ é ordenado em ordem crescente de $\text{Min}(\cdot)$ (Lema 6), $\text{tag}(T)$ está também desqualificado para os seguintes candidatos a prefixos de AP_i .

Case 2.ii $\text{Min}(M) < \text{Min}^*(T)$. Há possibilidade de casamento, então $\text{tag}(T)$ é um rótulo válido de M . Entretanto, se $\text{Min}(M) < \text{Min}(T)$, então fica assegurado

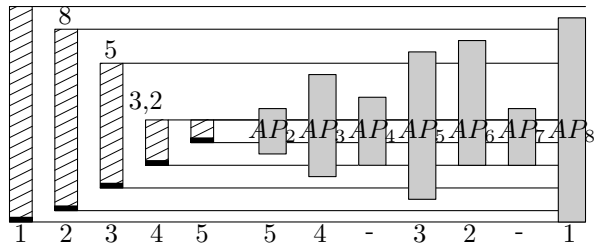


Figura 1.5. Representação gráfica dos resultados do procedimento de rotulamento (*tagging*). Consideramos o rotulamento de elementos de $PList(AP_1)$, representados com barras sombreadas à esquerda. As barras mais escuras (cor cinza) à direita representam os dados de outros processadores. Os número abaixo das barras representam os índices em $PList(AP_1)$ e $PTagList(1)$ (quando aplicável). O primeiro candidato a prefixo não tem rótulos devido ao caso 1 da Observação 1. O segundo recebe rótulo 8 baseado no caso 2.ii. O terceiro rejeita rótulos 6 e 8 baseados no caso 2.i, e é rotulado 5 baseado no caso 2.ii. O quarto recebe dois rótulos e bloqueia o rotulamento do quinto candidato a prefixo baseado no caso 2.ii.

que haverá um casamento entre M e um candidato a sufixo em $AP_{tag(T)}$, (a saber, N com $Max(N) = Max(T)$). Isso desqualifica todos os rótulos seguintes em $PTagList(i)$, pois eles levariam a seqüências que não podem ser maximais, uma vez que eles sobreporiam a $Pr1$ -subseqüência que certamente existe. Além disso, os candidatos a prefixos seguindo M em $PList(AP_i)$ levarão a seqüências que não podem ser A -maximais, assim eles não precisam ser rotulados.

Figura 1.5 ilustra o rotulamento de candidatos a prefixos e exemplifica os três casos acima. Algoritmo 3 contém o procedimento de rotulamento para os candidatos a prefixos de $PList(AP_i)$, chamados por simplicidade de Pl . $PTagList(i)$ é chamado de Tl por simplicidade e é pré-processado de acordo com a Afirmação 1.

Lema 15 Para $i \in [1, p]$ é possível rotular todos os elementos de $PList(AP_i)$ e $SList(AP_i)$, baseado nos valores de $Max(AP_j)$ e $Min(AP_j)$ para todo $j \in [1, p]$. Cada rótulo indica qual processador pode conter um casamento para um determinado candidato. Cada candidato é rotulado no máximo uma vez, com no

máximo duas exceções por processador. O tempo necessário é $O(|A|/p)$ e o espaço necessário é $O(p)$.

Prova: A prova é baseada no Algoritmo 3 e na Observação 1. Primeiro mostramos que Algoritmo 3 rotula corretamente todos os candidatos a prefixos pela demonstração da seguinte afirmação invariante: no momento do teste do laço, todos os elementos de $PList(AP_i)$ com índice menor que i_p receberam todos os rótulos que podem ser recebidos, e todos os elementos de $PtagList(i)$ com índice menor que i_t foram usados para rotular todos os possíveis candidatos.

A afirmação é obviamente verdadeira na primeira vez em que o teste é realizado. Suponha a afirmação verdadeira no início de uma iteração do laço. Três casos podem ocorrer:

- linha 4 é executada, pois $Max(Pl[i_p]) \geq Max(Tl[i_t])$ e caso 1 se aplica. Não mais rotulamentos podem ser aplicados a $Pl[i_p]$, e assim i_p é incrementado e a invariante permanece verdadeira.
- linha 6 é executada. Caso 2.i se aplica, então $Tl[i_t]$ não pode ser usado para rotular qualquer candidato a prefixo, permitindo o incremento de i_t .
- linha 8 é executada. Caso 2.ii se aplica e $Pl[i_p]$ é rotulado com $tag(Tl[i_t])$. Além disso, se $Min(Pl[i_p]) < Min(Tl[i_t])$, então, ainda devido ao caso 2.ii, o laço termina e não mais rotulamentos são feitos.

Algorithm 3 Rotulando (*Tagging*) uma Lista de Candidatos a Prefixos

Require: Listas Pl e Tl , com $|Pl|$ e $|Tl|$ elementos, respectivamente.

Ensure: Rotulamento dos elementos de Pl .

```

1:  $i_p \leftarrow 1, i_t \leftarrow 1, f \leftarrow false$ 
2: while  $i_p \leq |Pl|$  and  $i_t \leq |Tl|$  and not  $f$  do
3:   if  $Max(Pl[i_p]) \geq Max(Tl[i_t])$  then
4:      $i_p \leftarrow i_p + 1$ 
5:   else if  $Min(Pl[i_p]) \geq Min(Tl[i_t])$  then
6:      $i_t \leftarrow i_t + 1$ 
7:   else
8:     tag  $Pl[i_p]$  with  $tag(Tl[i_t])$ 
9:     if  $Min(Pl[i_p]) < Min(Tl[i_t])$  then
10:       $f \leftarrow true$ 
11:     end if
12:   end if
13: end while

```

Portanto, Algoritmo 3 executa o rotulamento corretamente. O tempo requerido é $O(|Pl| + |Tl|) = O(|A|/p + p) = O(|A|/p)$, incluindo o tempo para construir $PtagList(i)$. O espaço necessário é $O(p)$, para $PtagList(i)$. Um procedimento semelhante pode ser feito para $SList(AP_i)$.

Suponha agora que um candidato a prefixo $Pl[i'_p]$ é rotulado duas vezes, baseado em $Tl[i'_t]$ e $Tl[i'_t + 1]$. (É fácil de provar o fato de que os rótulos são consecutivos.) O rotulamento ocorre somente na linha 8. Para o segundo rótulo, fica claro que $Min(Pl[i'_p]) < Min(Tl[i'_t + 1])$, uma vez que o primeiro rotulamento ocorre, pois $Min(Pl[i'_p]) < Min*(Tl[i'_t]) \leq Min(Tl[i'_t + 1])$. Assim, quando um candidato a prefixo recebe o segundo rótulo o laço termina. Algo semelhante pode ocorrer a um candidato a sufixo, dando origem à segunda exceção da regra “um rótulo só”.

□

1.2.9. Obtendo Pr1-subseqüências entre-processadores

Depois do procedimento de rotulamento descrito na seção anterior, cada candidato a prefixo/sufixo pode ser associado a dois outros processadores: aquele que o contém e o especificado no rótulo. Alguns candidatos não têm rótulos e podem ser ignorados. Alguns poucos candidatos têm dois rótulos e têm que ser duplicados para a próxima fase.

A próxima fase envolve a verificação da existência de Pr1-subseqüências entre-processadores de A , isto é, Pr1-subseqüências que começam dentro de AP_i e terminam dentro de AP_j para algum par (i, j) , $1 \leq i < j \leq p$. Isso é feito pela verificação dos elementos de $PList(AP_i)$ que são rotulados com j e elementos de $SList(AP_j)$ que são rotulados com i . Estes elementos devem estar na memória local de um só processador para verificação pelo Algoritmo 2. A regra para escolher qual processador faz a verificação é simples: aquele cuja lista de candidatos é maior recebe os dados do outro processador. Caso ambas as listas tenham o mesmo tamanho, uma regra determinística qualquer é usada para desempatar. Por exemplo, uma regra pode ser: se $i + j$ é par, então P_i faz o trabalho de verificação, senão P_j o faz.

Isso pode ser feito para todos os pares de processadores usando duas rodadas de comunicação. Na primeira rodada, cada processador P_i envia para o processador $P_j \neq P_i$ o número de rótulos j que foram usados em $PList(AP_i)$ ou $SList(AP_i)$. Cada processador envia um número para cada um dos outros processadores, assim o tamanho da rodada de comunicação é $O(p)$. Com esses dados, cada par de processadores entra em acordo sobre qual processador deve receber os dados do outro. Esses dados são enviados numa próxima rodada de comunicação.

Note que cada processador pode enviar/receber no máximo o número de dados presentes nas suas próprias listas de candidatos a prefixos/sufixos. Portanto, esta rodada de comunicação tem tamanho $O(|A|/p)$.

Cada processador então busca por Pr1-subseqüências que começam ou terminam dentro da sua subseqüência local de A . Consideremos a busca pela Pr1-subseqüências que começa dentro de AP_i , $i \in [1, p[$. Para cada $j \in]i, p]$, processador P_i usa Algoritmo 2 para achar a maior Pr1-subseqüência com extremidades em AP_i e AP_j (supondo que este processador, e não P_j , foi selecionado para fazer o trabalho). Os candidatos a prefixos são os elementos

de $PList(AP_i)$ que foram rotulados com j (digamos que há r tais elementos) e os candidatos a sufixos que foram enviados pelo processador P_j (s elementos). Pelo Lema 13 o tempo requerido é $O(r+s)$. Este procedimento deve ser repetido pelo processador P_i para todo $j \in]i, p]$, levando um tempo total de $O(|PList(AP_i)|)$. Um procedimento similar deve ser feito por P_i para todo $j \in [1, i[$, levando tempo total $O(|SList(AP_i)|)$. O procedimento inteiro leva tempo $O(|A|/p)$.

Quando procurando por Pr1-subseqüências que começam dentro de AP_i , processador P_i deve primeiro buscar as seqüências que terminam em AP_p e prosseguem descendo até AP_{i+1} . Quando uma Pr1-subseqüência é achada o procedimento pode parar, pois as próximas Pr1-subseqüências estariam contidas na primeira. Isso significa que cada processador irá achar no máximo duas novas Pr1-subseqüências, uma terminando e uma começando na sua subseqüência local de A .

Podemos agora apresentar o lema seguinte, já provada pela discussão acima.

Lema 16 *Depois de rotular os candidatos a prefixos e sufixos como explicado na Seção 1.2.8, todas as Pr1-subseqüências entre-processadores que podem ser A-maximais podem ser achadas em tempo e espaço de $O(|A|/p)$ e com duas rodadas de comunicação de tamanhos $O(p)$ e $O(|A|/p)$. O número de seqüências é no máximo $2p$.*

Note que algumas das novas Pr1-subseqüências podem não possuir a Propriedade Pr2. O importante é que o procedimento descrito não perca nenhuma A-maximal possível. O próximo passo é achar as Pr1-subseqüências que são realmente A-maximais.

1.2.10. Obtenção das novas A-maximais

No passo final, todos os processadores enviam informações sobre as novas Pr1-subseqüências encontradas. Isso envolve uma quarta rodada de comunicação, de tamanho $O(p)$ (cada processador envia no máximo duas novas subseqüências e recebe todas as subseqüências enviadas pelos outros processadores). Cada processador então elimina a Pr1-subseqüência que está contida em uma outra Pr1-subseqüência. Isso pode ser redundante, mas é melhor fazer todos os processadores fazerem esta computação do que gastar uma outra rodada de comunicação.

Note que o procedimento descrito na seção anterior não gera duas Pr1-subseqüências que se sobrepõem, a menos que uma seja contida na outra. O motivo é que se duas Pr1-subseqüências se sobrepõem, então a união delas (isto é, a subseqüência de menor comprimento que contém ambas) também é uma Pr1-subseqüência. Isso é facilmente provado usando o Lema 1. A união é maior que cada uma das duas subseqüências que se sobrepõem, e assim deve ter sido detectada pelo procedimento descrito antes.

Algorithm 4 Eliminação de Pr1-subseqüências que não são A -maximais

Require: Lista L (com $|L|$ elementos) de pares de processadores para os quais há uma Pr1-subseqüência entre-processador.

Ensure: Lista N (com n elementos) de pares de processadores para os quais há uma A -maximal entre-processador.

```

1: for  $k \leftarrow 1$  to  $p$  do
2:    $V[k] \leftarrow k$ 
3: end for
4: for  $k \leftarrow 1$  to  $|L|$  do
5:    $i \leftarrow$  menor componente de  $L[k]$ 
6:    $j \leftarrow$  maior componente de  $L[k]$ 
7:   if  $j > V[i]$  then
8:      $V[i] \leftarrow j$ 
9:   end if
10: end for
11:  $n \leftarrow 0, k \leftarrow 1$ 
12: while  $k < p$  do
13:   if  $V[k] > k$  then
14:      $n \leftarrow n + 1$ 
15:      $N[n] \leftarrow (k, V[k])$ 
16:      $k \leftarrow V[k]$ 
17:   else
18:      $k \leftarrow k + 1$ 
19:   end if
20: end while

```

Cada Pr1-subseqüência está relacionada a um diferente par de processadores. O que deve ser verificado é quais pares geraram novas Pr1-subseqüências. Algoritmo 4 faz esta verificação.

Lema 17 *Algoritmo 4 acha todas as A -maximais entre-processadores baseado na lista de Pr1-subseqüências entre-processadores citada no Lema 16, em tempo e espaço $O(p)$.*

Prova: Linhas 1 a 10 constroem um array V em tempo $O(p + |L|) = O(p)$. $V[i]$, $i \in [1, p]$ armazena o maior j , para o que há uma nova Pr1-subseqüência que começa dentro de AP_i e termina dentro de AP_j . Se não há tal Pr1-subseqüência, então $V[i] = i$.

As linhas seguintes constroem a lista de novas A -maximais N . Uma afirmação invariante é que na linha 12 todas as novas A -maximais que começam dentro de AP_i para qualquer $i \in [1, k[$ foram já achadas. Isso certamente é verdadeiro para a primeira vez que a linha 12 é executada. No corpo do laço, se o teste na linha 13 resultar em *true*, então uma nova A maximal é achada. Todas

as Pr1-seqüências que começam dentro de AP_i para $k < i < V[k]$ estão contidas nesta nova A -maximal e devem ser ignoradas, que é feito na linha 16. Se o teste resultar em *false*, não há nova A -maximal que começa dentro de AP_k ; assim k é incrementado, mantendo assim a veracidade da invariante.

□

Um passo final é feito localmente por cada processador. Examinando a lista de novas A -maximais, processador P_i pode verificar se há uma A -maximal que contém inteiramente a sua subsequência local AP_i , que significa que seu próprio conjunto local de maximais $MList(AP_i)$ deve ser descartado. Esta verificação pode ser feita em tempo $O(p)$. Se há uma A -maximal entre-processadores que começa ou termina dentro de AP_i , uma última varredura de $MList(AP_i)$ pode eliminar as maximais locais que estão contidas em uma A -maximal maior. Esta varredura final pode ser feita em tempo $O(\log(|A|/p))$ se $MList(AP_i)$ é mantido em um *array* usado como um *buffer* circular e ordenado de acordo com a posição das maximais. Isso está de acordo com o Algoritmo 1.

Assim, podemos afirmar o seguinte.

Teorema 3 *Usando um CGM (Coarse-Grained Multicomputer) com p processadores, todas as subsequências maximais de uma seqüência A (já distribuída nas p memórias locais) podem ser achadas em tempo $O(|A|/p)$, usando $O(|A|/p)$ espaço local e $O(1)$ rodada de comunicação.*

Prova: A prova está baseada nos resultados desta seção, juntamente com os Lemas 11, 13, 15 e 16. Note que somente quatro rodadas de comunicação são necessárias, três das quais de tamanho $O(p)$ e uma de tamanho $O(|A|/p)$.

□

As conclusões sobre a solução deste problema serão dadas na Seção 1.4, juntamente com as sobre a solução do segundo problema.

1.3. Problema de subsequência comum mais longa

Dadas as cadeias X e Y de comprimentos m e n , respectivamente, o problema de *toda-subcadeia subsequência comum mais longa* (ou *all-substring longest common subsequence - ALCS*) acha os comprimentos das subsequências comuns mais longas entre X e *qualquer* subcadeia de Y . Uma breve introdução já foi feita na Seção 1.1.2. Iremos apresentar agora os detalhes para obter uma solução paralela para este problema.

1.3.1. O grafo orientado acíclico em grade (GDAG)

Assim como no problema de edição de cadeias [9, 11, 32], o problema de toda-subcadeia subsequência comum mais longa (ALCS) pode ser modelado por um grafo orientado acíclico em grade (*grid directed acyclic graph - GDAG*). Considere duas cadeias X e Y de comprimentos m e n , respectivamente. Para

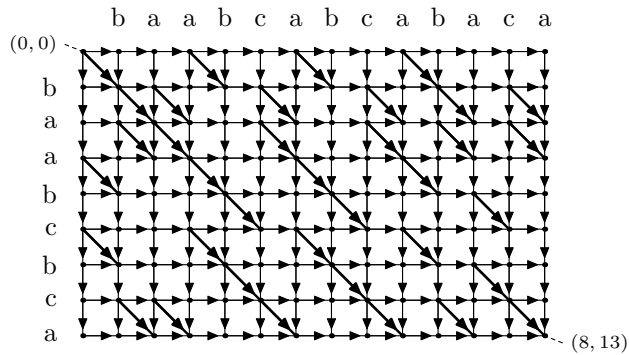


Figura 1.6. GDAG para o problema ALCS, com $X = \text{ba-abc}bcba$ e $Y = \text{baabc}cabaca$. As arestas diagonais mais escuras têm peso 1 e as demais arestas peso 0.

ilustrar as idéias neste texto, usaremos o exemplo seguinte. Sejam $X = \text{ba-abc}bcba$ e $Y = \text{baabc}cabaca$. O GDAG correspondente tem $(m+1) \times (n+1)$ vértices (ver Figura 1.6). Numeramos as linhas e colunas do GDAG começando com 0. As arestas do GDAG possuem pesos, definidos como se segue. Todas as arestas verticais e horizontais têm peso 0. A aresta do vértice $(i-1, j-1)$ ao vértice (i, j) tem peso 1 se $x_i = y_j$. Se $x_i \neq y_j$, a aresta tem peso 0 e pode ser ignorada (ou omitida no grafo).

Os vértices da linha do topo (superior) de G serão denotados por $T_G(i)$, e aqueles da linha de baixo (inferior) de G por $B_G(i)$, $0 \leq i \leq n$. Dada uma cadeia Y de comprimento n com símbolos y_1 a y_n , denote por Y_i^j a subcadeia de Y consistindo de símbolos y_i a y_j .

Definimos agora o *custo* ou *peso total* de um caminho entre dois vértices.

Definição 5 (Matriz C_G) Para $0 \leq i \leq j \leq n$, $C_G(i, j)$ é o custo ou peso total do melhor caminho entre vértices $T_G(i)$ e $B_G(j)$, representando o comprimento da subsequência comum mais longa entre X e a subcadeia Y_{i+1}^j . Se $i \geq j$ (Y_{i+1}^j é vazia ou não-existente), $C_G(i, j) = 0$.

Valores de $C_G(i, j)$ são mostrados na Figura 1.7. Por exemplo, $C_G(0, 9) = 8$. Isso significa que o comprimento da subsequência comum mais longa entre $X = \text{baabc}bcba$ e $Y_1^9 = \text{baabc}cabca$ é 8. Entretanto, note que $C_G(0, 10)$ também é 8. Isto é, se tomarmos um símbolo a mais de Y , o comprimento da subsequência comum mais longa ainda é o mesmo. Isso leva à definição seguinte do conceito de D_G , que trata dessa posição mais à esquerda (no exemplo, 9 e não 10) para se chegar a um valor de comprimento fixo (no exemplo 8).

Os valores de $C_G(i, j)$ possuem a seguinte propriedade. Para um i fixo, os valores de $C_G(i, j)$ com $0 \leq j \leq n$ formam uma seqüência não-decrescente que

$C_G(i, j)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	1	2	3	4	5	6	6	7	8	8	8	8	8
1	0	0	1	2	3	4	5	5	6	7	7	7	7	7
2	0	0	0	1	2	3	4	4	5	6	6	6	6	7
3	0	0	0	0	1	2	3	3	4	5	5	6	6	7
4	0	0	0	0	0	1	2	2	3	4	4	5	5	6
5	0	0	0	0	0	0	1	2	3	4	4	5	5	6
6	0	0	0	0	0	0	0	1	2	3	3	4	4	5
7	0	0	0	0	0	0	0	0	1	2	2	3	3	4
8	0	0	0	0	0	0	0	0	0	1	2	3	3	4
9	0	0	0	0	0	0	0	0	0	0	1	2	3	4
10	0	0	0	0	0	0	0	0	0	0	0	1	2	3
11	0	0	0	0	0	0	0	0	0	0	0	0	1	2
12	0	0	0	0	0	0	0	0	0	0	0	0	0	1
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 1.7. C_G correspondente ao GDAG da Figura 1.6

pode ser dada de forma implícita por apenas um desses valores de j para o qual $C_G(i, j) > C_G(i, j-1)$. Este fato foi usado por vários algoritmos seqüenciais para o problema LCS [10, 20, 29] e no algoritmo PRAM apresentado em [25], que é a base do nosso algoritmo paralelo e da definição seguinte.

Definição 6 (Matriz D_G) Seja G o GDAG para o problema ALCS para as cadeias X e Y . Para $0 \leq i \leq n$, $D_G(i, 0) = i$ e para $1 \leq k \leq m$, $D_G(i, k)$ indica o valor de j , tal que $C_G(i, j) = k$ e $C_G(i, j-1) = k-1$. Se não há tal valor, então $D_G(i, k) = \infty$.

Implícito nesta definição está o fato de que $C(i, j) \leq m$. Por conveniência, definimos D_G como uma matriz com índices começando com 0. Denotamos por D_G^i a linha i de D_G , isto é, o vetor linha formado por $D_G(i, 0), D_G(i, 1), \dots, D_G(i, m)$. Como exemplo, consideramos novamente o GDAG da Figura 1.6. Os valores de $D_G(i, k)$ estão mostrados na Figura 1.8.

O algoritmo proposto lida diretamente com esta representação. Para entender a matriz D_G , considere $D_G(i, k)$. O índice i é o índice inicial da cadeia Y na linha de topo de G . O valor k é o comprimento desejado da subsequência comum entre X e a subcadeia Y começando em i . Considere o GDAG da Figura 1.6. Se começarmos da posição i da linha de topo e seguirmos para a linha de baixo a uma posição dada por $D_G(i, k)$, então podemos obter um caminho de peso total k . De fato, $D_G(i, k)$ fornece a posição mais à esquerda que dá o peso total k . Ilustraremos isso com um exemplo. Como o comprimento da cadeia X é 8, o valor máximo que podemos esperar para k é, portanto, 8. $D_G(0, 8) = 9$. Isso significa o seguinte: no GDAG da Figura 1.6, começa com o índice 0 da linha de topo e pega arestas em qualquer uma das três direções:

$D_G(i, j)$	0	1	2	3	4	5	6	7	8
0	0	1	2	3	4	5	6	8	9
1	1	2	3	4	5	6	8	9	∞
2	2	3	4	5	6	8	9	13	∞
3	3	4	5	6	8	9	11	13	∞
4	4	5	6	8	9	11	13	∞	∞
5	5	6	7	8	9	11	13	∞	∞
6	6	7	8	9	11	13	∞	∞	∞
7	7	8	9	11	13	∞	∞	∞	∞
8	8	9	10	11	13	∞	∞	∞	∞
9	9	10	11	12	13	∞	∞	∞	∞
10	10	11	12	13	∞	∞	∞	∞	∞
11	11	12	13	∞	∞	∞	∞	∞	∞
12	12	13	∞	∞	∞	∞	∞	∞	∞
13	13	∞	∞	∞	∞	∞	∞	∞	∞

Figura 1.8. D_G correspondente ao GDAG da Figura 1.6.

pegando a diagonal obtemos um peso 1, enquanto que pegando as arestas horizontal ou vertical obtemos um peso 0. Agora, se desejamos obter um peso total 8, então a posição mais à esquerda na linha de baixo será 9. Assim, temos $D_G(0, 8) = 9$. Se fizermos i maior que 0, então comparamos X com a cadeia Y começando na posição i .

A propriedade seguinte foi provada em [25] e é importante para nossos resultados. Esta propriedade sugere a definição do conceito de V_G .

Propriedade 1 Para $0 \leq i \leq n-1$, linha D_G^{i+1} pode ser obtida da linha D_G^i pela remoção do primeiro elemento ($D_G^i(0) = i$) e a inserção de apenas um novo elemento (que pode ser ∞).

Definição 7 (Vetor V_G) Para $1 \leq i \leq n$, $V_G(i)$ é o valor do elemento finito que está presente na linha D_G^i , mas não está presente na linha D_G^{i-1} . Se tal elemento finito não existir, então $V_G(i) = \infty$.

Figura 1.9 mostra V_G para o GDAG da Figura 1.6. Assim, temos uma maneira econômica de armazenar D_G usando apenas espaço $O(m+n)$. Precisamos apenas armazenar a primeira linha de D_G , i.e., D_G^0 , de tamanho $O(m)$ e um vetor V_G de tamanho $O(n)$.

k	1	2	3	4	5	6	7	8	9	10	11	12	13
$V_G(k)$	∞	13	11	∞	7	∞	∞	10	12	∞	∞	∞	∞

Figura 1.9. V_G correspondente ao GDAG da Figura 1.6

Apresentamos o resultado seguinte sem prova. Detalhes podem ser encontrados em [2].

Teorema 4 *Dadas duas cadeias X e Y de comprimentos m e n , respectivamente, é possível resolver o problema ALCS seqüencialmente em tempo $O(mn)$ e espaço $O(n)$.*

O algoritmo seqüencial para ALCS é importante, pois ele será usado no algoritmo paralelo em cada processador, como será visto.

1.3.2. A estratégia básica do algoritmo BSP/CGM

Apresentamos agora o algoritmo BSP/CGM para o problema ALCS, dadas duas cadeias X e Y de comprimentos m e n , respectivamente. Por simplicidade, consideramos que o número de processadores p seja uma potência de 2 e m um múltiplo de p .

O algoritmo divide a cadeia X em p subcadeias de comprimento m/p que não se sobrepõem. O GDAG do problema original é dividido horizontalmente em *pedaços*, dando origem a p GDAGs de $m/p + 1$ linhas cada. Dois pedaços consecutivos compartilham uma linha em comum. Para $0 \leq i < p$, processador P_i resolve seqüencialmente o problema ALCS para as cadeias $X_{mi/p+1}^{m(i+1)/p}$ e Y , e computa o valor local de D_G . Do Teorema 4, o tempo necessário para os p processadores resolverem o subproblema ALCS em paralelo é $O(mn/p)$.

Usamos $\log p$ rodadas para juntar os resultados, nas quais pares de resultados parciais (para dois pedaços vizinhos) são juntados para dar origem a uma solução única com a união dos dois pedaços. A cada passo de junção, o número de processadores associado a cada pedaço é dobrado. Após $\log p$ rodadas, temos a solução do problema original. A soma dos tempos de todos os passos de junção é $O(n\sqrt{m}(1 + \log m/\sqrt{p}))$, como será visto. A Figura 1.10 ilustra o processo de junção.

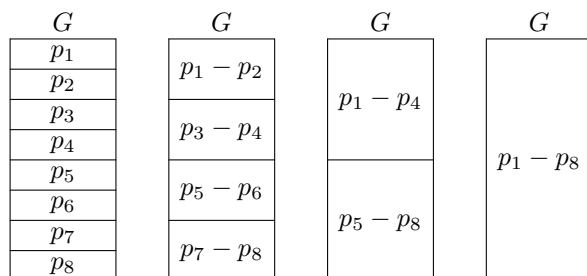


Figura 1.10. Juntando as soluções parciais de ALCS, com $p = 8$. Em cada pedaço do G indicamos os processadores usados na solução do GDAG do pedaço.

Assim, o algoritmo BSP/CGM para ALCS consiste de duas fases:

1. Cada um dos p processadores executa o algoritmo seqüencial ALCS para os seus *pedaços* locais e computa o D_G local.
2. Em $\log p$ rodadas pares de soluções contíguas são juntadas sucessivamente para obter a solução do problema original.

A parte mais difícil do algoritmo envolve a união ou junção de dois pedaços contíguos. Em particular, precisamos prestar especial atenção no armazenamento de D_G usando uma estrutura de dado compacta. Com isso, também diminuímos o tamanho das mensagens para serem comunicadas entre os processadores em cada rodada de junção. Isso será visto na próxima seção. O operador de junção é apresentado na Seção 1.3.4.

1.3.3. Estrutura de dados compacta para D_G

Para um GDAG G de tamanho $(n+1) \times (m'+1)$, as duas representações de D_G vistas até aqui não são adequadas. A representação *direta* em forma de matrizes (como na Figura 1.8) apresenta muita redundância e ocupa muito espaço, de $O(m'n)$, embora o acesso a qualquer valor individual de $D_G(i,k)$, dados i e k , possa ser feito em tempo $O(1)$. Por outro lado, a representação *incremental* através do uso dos vetores D_G^0 e V_G usa somente espaço $O(m'+n)$, mas não permite acesso rápido de valores individuais de D_G .

Definimos agora uma representação *compacta* de D_G , que ocupa espaço $O(n\sqrt{m'})$ e permite leituras de valores individuais de D_G em tempo $O(1)$. A estrutura pode ser construída de D_G^0 e V_G em tempo $O(n\sqrt{m'})$. Essa estrutura será essencial para nosso algoritmo. A construção é incremental pela adição de uma linha de D_G cada vez. Os valores de D_G são armazenados em um vetor chamado RD_G (D_G reduzido) de tamanho $O(nl)$, onde $l = \lceil \sqrt{m'+1} \rceil$.

Antes da descrição da construção de RD_G , damos uma idéia de como representar uma linha de D_G . Linha D_G^i ($0 \leq i \leq n$), de tamanho $m'+1$, é dividida em no máximo l sub-vetores, todos de tamanho l com a possível exceção do último. Estes sub-vetores são armazenados em posições separadas de RD_G , e precisamos de um vetor adicional de tamanho l para indicar a localização de cada sub-vetor. Este vetor adicional é denotado por Loc^i . A matriz $(n+1) \times l$ formada por todos os vetores Loc^i (um para cada D_G^i) é chamada Loc . Os índices de Loc começam em 0 (ver Figura 1.11). Pode-se mostrar facilmente que o valor de $D_G(i,k)$ pode ser obtido com Loc e RD_G em tempo $O(1)$.

Mostramos agora a construção de RD_G e Loc . Primeiro consideramos a inclusão da primeira linha D_G^0 . Cada sub-vetor de D_G^0 de tamanho l é alocado em um fragmento de RD_G de tamanho $2l$. O espaço extra será usado para alocar as próximas linhas de D_G . Assim, cada sub-vetor de D_G^0 é seguido por um espaço vazio de tamanho l . Como teremos bastante redundâncias, a inclusão da próxima linha de D_G pode ser feita de uma forma inteligente. O vetor RD_G (juntamente com Loc) pode conter todos os dados de D_G em apenas $O(nl)$ espaço, devido ao fato de que os sub-vetores de diferentes linhas de D_G podem se sobrepor. Com D_G^i já presente, para incluir D_G^{i+1} , podemos usar a maior parte dos

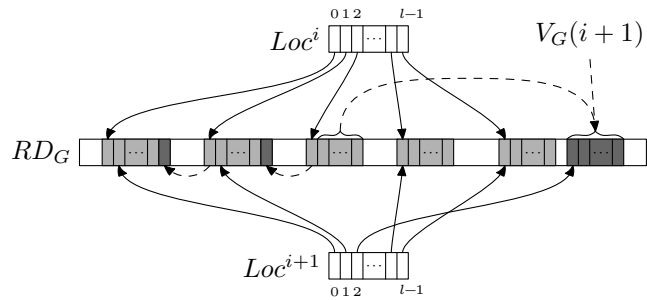


Figura 1.11. Armazenando D_G^i e D_G^{i+1} em RD_G

dados que já estão na estrutura de dados. Mais precisamente, a diferença é que D_G^{i+1} não possui o valor $D_G^i(0) = i$, mas pode ter um novo valor dado por $V_G(i+1)$. A inclusão de D_G^{i+1} (ver Figura 1.11) consiste de:

1. Determine o sub-vetor de D_G^i para inserir $V_G(i+1)$. Seja v o índice deste sub-vetor.
2. Determine a posição neste sub-vetor para inserir $V_G(i+1)$.
3. Todos os sub-vetores de D_G^{i+1} de índice *maior* que v são iguais àqueles de D_G^i , assim já presentes em RD_G . Basta fazer $Loc(i+1, j) = Loc(i, j)$ for $v < j < l$.
4. Todos os sub-vetores de D_G^{i+1} de índice *menores* que v são iguais àqueles de D_G^i , a menos de um deslocamento para a esquerda e da inclusão de um novo elemento à direita (precisamente o elemento *jogado fora* do próximo sub-vetor). Os sub-vetores podem ser deslocados para a esquerda facilmente fazendo $Loc(i+1, j) = Loc(i, j) + 1$ para $0 \leq j < v$. O novo elemento de cada sub-vetor pode ser escrito imediatamente à direita do sub-vetor, dado que há espaço vazio para isso em RD_G . Caso contrário alocamos um novo fragmento de tamanho $2l$.
5. O sub-vetor de índice v é modificado de tal modo que um novo sub-vetor deve ser alocado em RD_G , com a inclusão de $V_G(i+1)$. $Loc(i+1, v)$ indica a posição deste novo sub-vetor.

Na Figura 1.11 o elemento $V_G(i+1)$ deve ser inserido no sub-vetor de índice 2. As áreas mais escuras representam os dados escritos em RD_G neste passo de inclusão. As setas pontilhadas indicam a cópia de dados. A inclusão de D_G^{i+1} envolve a determinação de uma nova linha de Loc (tempo e espaço $O(nl)$) e alguns novos sub-vetores em RD_G para passos 4 e 5 (tempo e espaço $O(nl)$ numa análise amortizada).

Resumimos os resultados desta seção no seguinte.

Teorema 5 Considere o GDAG G do problema ALCS para as cadeias X e Y . De D_G^0 e V_G podemos reconstruir a representação de D_G em tempo e espaço $O(n\sqrt{m'})$, de tal modo que qualquer valor de D_G possa ser acessado em tempo $O(1)$.

1.3.4. A operação básica de junção de duas soluções parciais

A estratégia apresentada na Seção 1.3.2 utiliza como operação básica a união ou junção de dois pedaços contíguos para formar um pedaço maior. Após a última operação de junção obtemos a matriz D_G correspondente ao GDAG original. Consideremos a junção de dois GDAGs U e L de $m' + 1$ linhas cada, resultando em um GDAG G de $2m' + 1$ linhas. Usamos o exemplo dado para ilustrar esta operação.

Seja D_U correspondente à metade superior do GDAG (primeiras 5 linhas do exemplo, da linha 0 até linha 4) e D_L correspondente à metade inferior do GDAG (linhas 4 até linha 8). D_U e D_L estão mostrados na Figura 1.12. Primeiro mostramos como obter D_G usando D_U e D_L . A idéia básica é a mesma usada em [25].

$D_U(i, j)$	0	1	2	3	4	$D_L(i, j)$	0	1	2	3	4
0	0	1	2	3	4	0	0	1	2	6	9
1	1	2	3	4	10	1	1	2	5	6	9
2	2	3	4	7	10	2	2	3	5	6	9 ^o
3	3	4	6	7	10	3	3	4	5	6 ^o	9 [•]
4	4	6	7	10	∞	4	4	5	6 ^o	8 [•]	9
5	5	6	7	10	∞	5	5	6	8	9	13
6	6	7	9	10	∞	6	6	7	8	9	13
7	7	9	10	13	∞	7	7	8	9	11	13
8	8	9	10	13	∞	8	8	9	11	13	∞
9	9	10	11	13	∞	9	9	10	11	13	∞
10	10	11	13	∞	∞	10	10	11 ^o	13 [•]	∞	∞
11	11	13	∞	∞	∞	11	11	12	13	∞	∞
12	12	13	∞	∞	∞	12	12	13	∞	∞	∞
13	13	∞	∞	∞	∞	13	13	∞	∞	∞	∞

Figura 1.12. D_U e D_L correspondentes à metade superior e à metade inferior do GDAG da Figura 1.6. O significado de \bullet e \circ será explicado adiante.

Note primeiro que $D_G^i(k) = D_G(i, k)$ representa o menor valor de j , tal que $C_G(i, j)$, o peso total do melhor caminho entre $T_G(i)$ (vértice i da linha de topo do GDAG G) e $B_G(j)$ (vértice j da linha de baixo do GDAG G), é k . Todos os caminhos de $T_G(i) = T_U(i)$ a $B_G(j) = B_L(j)$ têm que cruzar a fronteira $B_U = T_L$ em algum vértice, e o peso total do caminho é a soma dos pesos do intervalo

em U e em L . Assim, para determinar $D_G(i, k)$ precisamos considerar caminhos que cruzam U com peso total l e depois cruzam L com peso total $k - l$, para todo l de 0 a m' .

Tendo fixado um certo valor de l , o menor valor de j , tal que existe um caminho de $T_G(i)$ a $B_G(j)$ com peso l em U e peso $k - l$ em L , é dado por $D_L(D_U(i, l), k - l)$, uma vez que $D_U(i, l)$ é o primeiro vértice na fronteira que está a uma distância de l de $T_G(i)$ e $D_L(D_U(i, l), k - l)$ é o primeiro vértice que está a uma distância de $k - l$ do vértice na fronteira.

Pelas considerações acima, temos:

$$D_G(i, k) = \min_{0 \leq l \leq m'} \{D_L(D_U(i, l), k - l)\} \quad (1)$$

Observe que ao manter i fixo e variar k , as linhas de D_L usadas são sempre as mesmas. A variação de k só muda o elemento que deve ser consultado em cada linha. Para cada linha de D_L consultada, um elemento diferente é tomado, devido ao termo $-l$. Esta operação de deslocamento e a observação seguinte sugerem a seguinte definição de *shift*, *diag* e *MD*. Antes dessas definições consideremos a obtenção, digamos, de $D_G(1, 6)$ e $D_G(1, 5)$, usando a Equação 1.

$$D_G(1, 6) = \min_{0 \leq l \leq m'} \{D_L(D_U(1, l), 6 - l)\}$$

Isso envolve o mínimo dos valores marcados com \bullet na Figura 1.12, dando o valor 8. Por outro lado, para obter

$$D_G(1, 5) = \min_{0 \leq l \leq m'} \{D_L(D_U(1, l), 5 - l)\},$$

temos que computar o mínimo dos valores marcados com \circ , que é 6.

Note que se deslocarmos as linhas apropriadas de D_L (linhas 1, 2, 3, 4, 10) para a direita, com cada linha deslocada de uma posição para a direita com relação à linha anterior, todos os valores, cujo mínimo necessita ser computado, são alinhados convenientemente na mesma coluna, com todas as novas posições preenchidas com ∞ (Figura 1.13). Vemos que todos os valores mínimos de cada respectiva coluna dão a linha 1 inteira de D_G . A disposição da Figura 1.13 é denominada $MD[1](i, j)$, que será formalizada nas definições de *shift* e *diag*.

Definição 8 ($shift[l, W, c]$) Dado um vetor W de comprimento $s + 1$ (índices de 0 a s), para todo l ($0 \leq l \leq c - s$) defina $shift[l, W, c]$ como o vetor de comprimento $c + 1$ (índices de 0 a c) tal que:

$$shift[l, W, c](i) = \begin{cases} \infty & \text{if } 0 \leq i < l \\ W(i + l) & \text{if } l \leq i \leq s + l \\ \infty & \text{if } s + l < i \leq c \end{cases}$$

Em outras palavras, $shift[l, W, c]$ é o vetor W deslocado para a direita de l posições e completado com ∞ .

$MD[1](i, j)$	0	1	2	3	4	5	6	7	8
0	1	2	5	6	9	∞	∞	∞	∞
1	∞	2	3	5	6	9 $^\circ$	∞	∞	∞
2	∞	∞	3	4	5	6 $^\circ$	9 $^\bullet$	∞	∞
3	∞	∞	∞	4	5	6 $^\circ$	8 $^\bullet$	9	∞
4	∞	∞	∞	∞	10	11 $^\circ$	13 $^\bullet$	∞	∞
Minimum	\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	\downarrow
	1	2	3	4	5	6	8	9	∞

Figura 1.13. Note que as *bolinhas brancas e pretas estão todas alinhadas verticalmente*. $MD[1]$ pode ser usada para obter a linha 1 de D_G .

Com esta definição podemos reescrever a Equação 1:

$$D_G(i, k) = \min_{0 \leq l \leq m'} \{ \text{shift}[D_L^{D_U(i, l)}, l, 2m'](k) \} \quad (2)$$

Considerando todas as linhas relativas a um certo valor de i podemos obter a matriz $MD[i]$, através das seguintes definições, para achar todos os elementos de D_G^i .

Definição 9 ($Diag[W, M, l_0]$) *Seja W um vetor (começando com índice 0) de inteiros em ordem crescente, tal que os primeiros $\bar{m} + 1$ elementos são números finitos e $W(\bar{m}) \leq n$. Seja M uma matriz $(n + 1) \times (m' + 1)$ (índices começando com 0). $Diag[W, M, l_0]$ é uma matriz $(\bar{m} + 1) \times (2m' + 1)$, tal que sua linha de índice l é $\text{shift}[M^{W(l)}, l + l_0, 2m']$.*

$Diag[W, M, l_0]$ tem suas linhas copiadas de uma matriz M . A seleção de quais linhas são copiadas é feita pelo vetor W . Cada linha copiada é deslocada de uma coluna para a direita em relação à linha anterior. A quantidade para deslocar a primeira linha copiada é indicada por l_0 .

Definição 10 ($MD[i]$) *Seja G um GDAG para o problema ALCS, formado pela junção dos GDAGs U (upper) e L (lower). Então, $MD[G, i] = Diag[D_U^i, D_L, 0]$. Quando G está claro no contexto, usaremos apenas a notação $MD[i]$.*

A Figura 1.13 mostra $MD[1]$ que pode ser usado para obter D_G^1 . Assim, pela obtenção do mínimo de cada coluna de $MD[i]$ temos D_G^i . Se denotarmos por $Cmin[M]$ os mínimos das respectivas colunas da matriz M , então podemos escrever:

$$D_G^i(k) = Cmin[MD[i]](k) \quad (3)$$

A matriz $MD[i]$ pertence a uma classe de matrizes conhecidas como *matrizes totalmente monotônicas* (totally monotone matrices) [1]. Uma matriz é

chamada *monotônica* se o mínimo de uma coluna está abaixo ou à direita do mínimo de sua coluna vizinha direita. Se dois ou mais elementos têm o mínimo, então pegue o elemento superior. Um matriz é chamada *totalmente monotônica* se cada uma de suas 2×2 sub-matrizes é monotônica.

Antes de prosseguir, definimos a maneira para comparar os elementos quando aparecem valores ∞ em $MD[i]$.

Observação 2 *Dados dois valores ∞ de uma mesma coluna de $MD[i]$, $MD[i](l_1, j)$ e $MD[i](l_2, j)$, sendo $l_1 < l_2$, dizemos que $MD[i](l_1, j) > MD[i](l_2, j)$ se e somente se $l_2 < j$ (isto é, os dois elementos estão acima da diagonal principal.). Em caso contrário, dizemos que $MD[i](l_1, j) < MD[i](l_2, j)$.*

Vamos enunciar o seguinte sem provar. As provas podem ser encontradas em [2].

Teorema 6 *Fazendo as comparações de acordo com a Observação 2, a matriz $MD[i]$ é totalmente monotônica.*

Dado que todas as matrizes $MD[i]$ são totalmente monotônicas, a união ou junção de GDAGs pode ser feita por meio da busca do mínimo das colunas para todas as matrizes, por um algoritmo baseado em [1], que leva somente tempo $O(m)$ para cada uma das $n + 1$ matrizes (pois as matrizes têm altura m).

Mesmo com este algoritmo, entretanto, o tempo total é ainda $O(nm)$. Isso não é ainda satisfatório. Para resolver o problema de junção em tempo menor observamos que, dada a similaridade entre linhas adjacentes de D_G , matrizes $MD[i]$ são também semelhantes para valores próximos a i . Isso será explicado a seguir.

1.3.5. Eliminação da redundância

Exploramos a propriedade de que $r + 1$ linhas consecutivas de D_U são r -variantes [25], i.e., para obter qualquer linha a partir de qualquer outra linha, precisamos somente remover no máximo r elementos e inserir no máximo outros r elementos. Mais importante, com $r + 1$ linhas consecutivas de comprimento $m' + 1$, fica possível obter um vetor de elementos que são comuns em todas as $r + 1$ linhas de tamanho $m' + 1 - r$, que chamaremos de *vetor comum*. Os elementos do vetor comum podem ser não-contíguos. Usamos a notação $D_U^{i_0, r}$ para indicar o vetor comum da linha de $D_U^{i_0}$ a $D_U^{i_0+r}$ para um GDAG U . Usamos $r = \lceil \sqrt{m'} \rceil$.

Pode-se mostrar que todos os elementos de um vetor $D_U^{i_0}$ estão também presentes no vetor $D_U^{i_0+r}$, com exceção daqueles que são menores que $i_0 + r$. Por exemplo, considere D_U da Figura 1.12. Temos $m' = 4$ e $r = 2$. Os elementos comuns a $D_U^0 = (0, 1, 2, 3, 4)$ e $D_U^2 = (2, 3, 4, 7, 8)$ são 2, 3, 4 (os últimos elementos de D_G^0).

Pode-se também mostrar que todos os elementos presentes em ambos os vetores $D_U^{i_0}$ e $D_U^{i_0+r}$ estão também presentes em vetores D_U^i para $i_0 < i <$

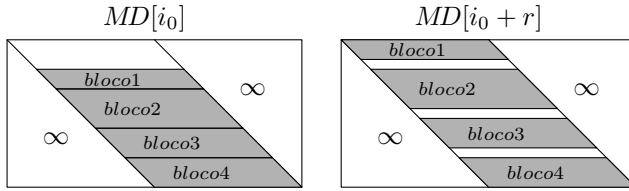


Figura 1.14. Estruturas das matrizes $MD[i_0]$ e $MD[i_0 + r]$, mostrando três blocos comuns

$i_0 + r$. Portanto, podemos ter uma forma simples de determinar o vetor comum para um grupo de linhas: precisamos simplesmente pegar D_U e ler todos os elementos de $D_U^{i_0}$, ignorando aqueles que são menores que $i_0 + r$. Isso leva tempo $O(m')$. No exemplo anterior, temos $D_U^{0,2} = (2, 3, 4)$.

Mais ainda, é importante determinar quais elementos são adjacentes em todas as linhas formando *pedaços indivisíveis*. Estes elementos serão denominados *fragmentos comuns*. No exemplo, pegando linhas 6, 7, 8, $D_U^{6,2} = (9, 10, \infty)$ e os fragmentos comuns são (9, 10) e (∞).

A determinação dos fragmentos comuns pode ser feita usando o vetor V_U . De $V_U(i_0 + 1)$ a $V_U(i_0 + r)$ temos os elementos que não aparecem no vetor comum e podem ser usados para dividi-lo em fragmentos comuns. Isso leva tempo $O(\log m')$ para cada elemento e um tempo total de $O(r \log m')$. Os fragmentos comuns são numerados de 0 a r e o fragmento t será denotado $D_U^{i_0, r}[t]$.

Agora voltamos ao processo de junção do algoritmo. Obtemos o vetor comum e os fragmentos comuns da matriz D_U . Ao invés de construir $MD[i]$ para cada individual i , usamos as linhas $D_U^{i_0}$ a $D_U^{i_0 + r}$ para construir matrizes $MD[i_0]$ a $MD[i_0 + r]$ e, como já mencionado, as similaridades entre linhas próximas umas das outras implicam similaridades entre matrizes $MD[i]$ para valores próximos de i . O vetor comum $D_U^{i_0, r}$ contém os índices das linhas de D_L que estão presentes em todas as matrizes de $MD[i_0]$ a $MD[i_0 + r]$. Cada fragmento $D_U^{i_0, r}[t]$ indica um conjunto de linhas de D_L , chamado *blocos comuns*, que podem ser usados em linhas *adjacentes* em todas estas matrizes.

Considere um fragmento comum $D_U^{i_0, r}[t]$, $0 \leq t \leq r$. Todas as matrizes $MD[i]$ com $i_0 \leq i \leq i_0 + r$ conterão $Diag[D_U^{i_0, r}[t], D_L, l_{i,t}]$ como um conjunto de linhas contíguas da linha $l_{i,t}$, onde $l_{i,t}$ varia de matriz a matriz e de bloco a bloco. Isso é ilustrado na Figura 1.14.

Em cada matriz é necessário resolver o problema dos mínimos das colunas. Podemos então evitar a computação repetitiva determinando primeiro os mínimos das colunas dos blocos comuns. Temos então a seguinte definição.

Definição 11 ($ContBl[i_0, r, t]$)

$$ContBl[i_0, r, t] = Cmin[Diag[D_U^{i_0, r}[t], D_L, 0]]$$

Em outras palavras, $ContBl[i_0, r, t]$ é um vetor dos mínimos das colunas do bloco t comum às matrizes $MD[i_0]$ a $MD[i_0 + r]$.

Considere agora a idéia de *contração de linha* de uma matriz (totalmente) monotônica.

Definição 12 [Contração de linhas de uma matriz (totalmente) monotônica] Seja M uma matriz (totalmente) monotônica. Uma *contração de linha* aplicada a um conjunto de linhas contíguas é a substituição de todas aquelas linhas em M por uma só linha. Os elementos da coluna i desta nova linha é o mínimo dos elementos presentes na coluna i das linhas originais substituídas.

Pode-se mostrar que após a contração a matriz continua sendo (totalmente) monotônica.

Se para cada matriz $MD[i]$ fazemos sucessivas *contrações de linhas*, uma para cada um dos blocos comuns, o resultado será uma que chamamos de $ContMD[i]$, tal que $Cmin[MD[i]] = Cmin[ContMD[i]]$.

A contração de cada bloco pode ser feita por um algoritmo apresentado em [1] adaptado para matrizes com poucas linhas. Um bloco t de m_t linhas é contraído em tempo $O(m_t \log(m'/m_t))$. Para cada linha i em um bloco particular, este algoritmo indica o intervalo de colunas que têm seus mínimos nesta coluna. Esta representação indireta dos mínimos das colunas pode ser consultada em tempo $O(\log m')$ para cada elemento de $ContBl[i_0, r, t]$. Na análise a seguir, o fator $\log m'$ na complexidade de tempo vem deste tempo de consulta. A contração de todos os blocos pode ser feita em tempo $O(m' \log m')$.

Os blocos comuns aparecem nas matrizes $MD[i]$ em diferentes posições, mas o resultado da busca pelos mínimos das colunas destes blocos pode ser usado em todas as matrizes. Mais precisamente, se o bloco comum t aparece começando com a linha $l_{i,t}$ da matriz $MD[i]$, a contração deste bloco na matriz é feita simplesmente pela substituição do bloco pelo vetor $shift[l_{i,t}, ContBl[i_0, r, t], 2m]$.

Além das $r + 1$ linhas obtidas da contração dos blocos comuns, cada matriz $ContMD[i]$ contém no máximo r linhas adicionais. Note que $r = \lceil \sqrt{m'} \rceil$. Portanto, a contração das matrizes reduz a altura dessas matrizes a $O(\sqrt{m'})$. A contração das matrizes $ContMD[i_0]$ a $ContMD[i_0 + r]$ pode ser feita por uma simples manipulação de ponteiros. De fato, para construir $ContMD[i + 1]$, podemos usar $ContMD[i]$, remover a primeira linha e inserir uma nova. Uma estrutura similar àquela para D_G da Seção 1.3.3 pode ser usada aqui.

Usamos o algoritmo de Aggarwal et al. [1] para achar todos os mínimos das colunas da primeira matriz do intervalo, $ContMD[i_0]$, obtendo linha $D_G^{i_0}$ em tempo $O(m' \log m')$. Para linha D_G^i , $i_0 < i \leq i_0 + r$, usando a Propriedade 1, só precisamos determinar $V_G(i)$ de D_G^{i-1} . Fazemos isso tomando de $ContMD[i]$ $O(\sqrt{m'})$ colunas, espaçadas de $\lceil \sqrt{m'} \rceil$ e achando seus mínimos em tempo $O(\sqrt{m'} \log m')$. Isso nos dá uma $\lceil \sqrt{m'} \rceil$ -amostra de D_G^i , que pode ser comparado a D_G^{i-1} para dar-nos um intervalo de tamanho $O(\sqrt{m'})$, onde $V_G(i)$ pode

ser obtido. Para achar $V_G(i)$ fazemos uma outra determinação dos mínimos das colunas em tempo $O(\sqrt{m'} \log m')$. Fazendo isso para r valores de i , gastamos tempo $O(r\sqrt{m'} \log m') = O(m' \log m')$. As linhas de D_G podem ser armazenadas na estrutura de dados descrita na Seção 1.3.3 para comparação de linhas adjacentes (os dados para linhas já usadas podem ser descartadas).

Como há um total de $(n+1)/(r+1)$ grupos de $r+1$ matrizes $MD[i]$ para processar, o tempo total para determinar D_G^0 e V_G de D_U a D_L é $O((nm'/r) \log m') = O(n\sqrt{m'} \log m')$. Podemos dividir este trabalho usando q processadores, dividindo D_U entre os processadores. Cada bloco de r linhas de D_U pode ser usado para determinar as r linhas de D_G , o processamento de cada bloco sendo independente dos outros blocos. Com isso, o tempo fica $O((n\sqrt{m'} \log m')/q)$.

Precisamos considerar o tempo para construir a representação de D_U e D_L , conforme mostrado antes. A representação de D_L deve estar disponível na memória local de todos os processadores. Esta construção leva tempo $O(n\sqrt{m'})$, ou $O(n\sqrt{m'}/q)$ usando q processadores. Temos assim o seguinte.

Lema 18 *Seja G' um GDAG $(2m'+1) \times (n+1)$ para o problema ALCS, formado pela junção dos GDAGS $(m'+1) \times (n+1)$ U (upper) e L (lower). A determinação de D_G^0 e V_G de D_U , V_U , D_L^0 e V_L pode ser feita por q processadores em tempo $O(n\sqrt{m'}(1 + \log m'/q))$ e espaço $O(n\sqrt{m'})$.*

1.3.6. Análise do algoritmo completo

Dadas as cadeias X e Y de comprimentos m e n , respectivamente, fase 1 do algoritmo BSP/CGM da Seção 1.3.2 resolve o problema ALCS para os p GDAGs definidos para as p subcadeias de X em tempo $O(mn/p)$.

Precisamos obter a complexidade de tempo da fase 2. Considere uma operação básica de união que junta um GDAG superior U' e um GDAG inferior L' , digamos de tamanhos $(m'+1) \times (n+1)$ cada, para produzir o GDAG união G' de tamanho $(2m'+1) \times (n+1)$. O valor de m' dobra em cada passo de união ou junção até o último passo em que temos a solução (quando $2m' = m$).

Quando há q processadores para lidar com o GDAG G' , temos $m' = mq/2p$. Pelo Lema 18, a complexidade de tempo fica:

$$O\left(n\sqrt{m'}\left(\sqrt{\frac{q}{2p}} + \frac{\log \frac{mq}{2p}}{\sqrt{2pq}}\right)\right) = O\left(\frac{n\sqrt{m}}{\sqrt{p}}\left(\sqrt{q} + \frac{\log m}{\sqrt{q}}\right)\right)$$

A quantidade de processadores q envolvidos na obtenção de cada GDAG dobra em cada processo de união. Assim, a soma dos tempos de todos os processos de união é:

$$O\left(\frac{n\sqrt{m}}{\sqrt{p}}\left(\sum_{i=1}^{\log p} (\sqrt{2})^i + \sum_{i=1}^{\log p} \frac{\log m}{(\sqrt{2})^i}\right)\right) = O\left(\frac{n\sqrt{m}}{\sqrt{p}}(\sqrt{p} + \log m)\right) = O\left(n\sqrt{m}\left(1 + \frac{\log m}{\sqrt{p}}\right)\right)$$

Para conseguir um ganho (*speed-up*) linear precisamos fazer este tempo $O(mn/p)$. Isso é conseguido se $p < \sqrt{m}$. O espaço requerido para o algoritmo proposto é $O(n\sqrt{m})$ por processador, devido à representação de D_L em cada processo de união.

Quanto a requisitos de comunicação, com q processadores fazendo a união, cada processador determina n/q elementos de V_G que precisam ser transmitidos a outros $2q - 1$ processadores para o próximo passo da união. Isso resulta em $O(n)$ dados transferidos por processador. O processador que determina D_G^0 precisa transferir os mq/p elementos deste vetor a outros $2q - 1$ processadores, resultando em uma rodada de comunicação onde $O(mq^2/p)$ dados são transmitidos.

Para alguma constante C , isso pode também ser feito em C rodadas de comunicação em que cada processador transmite $O(mq^{1+1/C}p)$ dados: na primeira rodada o processador que determinou D_G^0 , faz um *broadcast* deste vetor a $\lfloor q^{1/C} \rfloor$ outros processadores, que então transmite a $\lfloor q^{2/C} \rfloor$ outros processadores e assim por diante. O último passo de união se dá quando a maior quantidade de dados é transmitida por processador, $O(mp^{1/C} + n)$.

Concluimos esta seção com o seguinte principal resultado, que é um algoritmo BSP/CGM de ganho linear para o problema ALCS.

Teorema 7 *Dadas duas cadeias X e Y de comprimentos m e n , respectivamente, o problema ALCS pode ser resolvido por $p < \sqrt{m}$ processadores em tempo $O(mn/p)$, espaço por processador $O(n\sqrt{m})$, e $O(C \log p)$ rodadas de comunicação, para alguma constante escolhida C , em que $O(mp^{1/C} + n)$ dados são transmitidos de/para cada processador.*

1.4. Conclusão

Concluiremos este texto comentando as soluções paralelas obtidas para os dois problemas abordados.

Apresentamos um algoritmo paralelo que acha todas as subseqüências maximais de uma seqüência A com ganho (*speed-up*) linear e alta escalabilidade. O tamanho da rodada de comunicação, isto é, a quantidade de dados transmitidos numa rodada de comunicação, é limitado a $O(|A|/p)$. Conjecturamos que se $|A| \gg p$ o tamanho médio das rodadas de comunicação deve ser menor que $|A|/p$. De fato, por experimentos feitos com uma seqüência X de números aleatórios conjecturamos que o tamanho médio de $PList(X)$ seja $O(\log(|X|))$. O tempo de execução do algoritmo é dominado pelo tempo do primeiro passo, para achar as subseqüências maximais locais.

Não é trivial derivar este algoritmo paralelo de tempo $O(|A|/p)$ e espaço $O(|A|/p)$ por processador, e não é intuitivo que possa ser obtido um algoritmo paralelo que requer um número constante de rodadas de comunicação. Para isso, tivemos que explorar as propriedades das maximais locais que são potenciais candidatos para ser juntadas a fim de formar maximais maiores, além de um processo eficiente para juntar os candidatos a maximais locais. Por essa razão precisamos de muitos lemas auxiliares para se chegar ao resultado.

Algumas adaptações podem ser feitas a este algoritmo. Por exemplo, é fácil fazê-lo funcionar para uma seqüência *circular* de números, que pode ser importante quando se lida com cadeias circulares de nucleotídios. Além disso, se as

melhores k maximais são necessários, um algoritmo paralelo de seleção [31] pode ser usado para achar a k -ésima melhor maximal em tempo $O(|A|/p)$ e $O(\log p)$ rodadas de comunicação.

Para o segundo problema, mostramos que resolvendo o problema ALCS, resolvemos também o problema LCS. Mesmo considerando o problema mais geral, conseguimos apresentar um algoritmo paralelo BSP/CGM com p processadores de tempo $O(mn/p)$, onde m e n são os comprimentos das duas cadeias dadas, e $O(\log p)$ rodadas de comunicação. Do nosso conhecimento, este é o melhor algoritmo BSP/CGM conhecido na literatura para o problema ALCS.

Para derivar este algoritmo eficiente de ganho linear, exploramos as propriedades de matrizes totalmente monotônicas, as similaridades entre linhas da matriz D_G matriz e a similaridade entre matrizes consecutivas $MD[i]$. Isso possibilitou a redução do espaço necessário para armazenar informações essenciais, reduzindo como consequência a quantidade de dados transmitidos em cada rodada de comunicação.

Como problemas futuros, podemos buscar outros algoritmos paralelos, para estes mesmos problemas, com melhores complexidades de computação local e em números de rodadas de comunicação. Em particular, para o segundo problema, podemos propor novas estruturas de dados que armazenam as informações pertinentes que sejam eficientes, tanto em espaço, como em tempo de acesso às informações armazenadas. Nesse sentido, já há diversos trabalhos com resultados promissores [33, 8].

1.5. Agradecimentos

Este texto é baseado em vários trabalhos [2, 3, 5, 6, 7] publicados em co-autoria com os Professores Edson Norberto Cáceres e Carlos Eduardo Rodrigues Alves. Apresento meus sinceros agradecimentos a estes colaboradores. Sem a sua colaboração este texto não seria possível. Também quero agradecer aos revisores, que muito melhoraram este texto, com seus comentários e sugestões.

Referências bibliográficas

- [1] A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987.
- [2] C. E. R. Alves, E. N. Cáceres, and S. W. Song. Sequential and parallel algorithms for the all-substrings longest common subsequence problem. Technical report, Universidade de São Paulo, October 2002.
- [3] C. E. R. Alves, E. N. Cáceres, and S. W. Song. A (bsp/cgm) algorithm for the all-substrings longest common subsequence problem. In *Proceedings 17th IEEE Annual International Parallel & Distributed Processing Symposium (IPDPS 2003)*, pages 22–26. IEEE Computer Society, 2003.
- [4] C. E. R. Alves, E. N. Cáceres, and S. W. Song. BSP/CGM algorithms for maximum subsequence and maximum subarray. In *Proceedings Euro PVM/MPI 2004 - 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 139–146. Springer Verlag, 2004.
- [5] C. E. R. Alves, E. N. Cáceres, and S. W. Song. A BSP/CGM algorithm for finding all maximal contiguous subsequences of a sequence of numbers. Technical report, Universidade de São Paulo, January 2005.
- [6] C. E. R. Alves, E. N. Cáceres, and S. W. Song. A BSP/CGM algorithm for finding all maximal contiguous subsequences of a sequence of numbers. In *Proceedings Euro-Par 2006*, volume 4128 of *Lecture Notes in Computer Science*, pages 831–840. Springer Verlag, 2006.
- [7] C. E. R. Alves, E. N. Cáceres, and S. W. Song. A coarse-grained parallel algorithm for the all-substrings longest common subsequence problem. *Algorithmica*, 45:301–335, 2006.
- [8] C. E. R. Alves, E. N. Cáceres, and S. W. Song. Efficient representation of row-sorted 1-variant matrices for parallel string applications. In *Proceedings 7th International Conference on Algorithms and Architectures for Parallel Processing*, Lecture Notes in Computer Science. Springer Verlag, 2007. Accepted.
- [9] C.E.R. Alves, E. N. Cáceres, F. Dehne, and S. W. Song. Parallel dynamic programming for solving the string editing problem on a CGM/BSP. In *Proceedings SPAA'02 - 14th ACM Symposium on Parallel Algorithms and Architectures*, page accepted. ACM PRESS, 2002.
- [10] A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2:315–336, 1987.
- [11] A. Apostolico, L.L. Larmore M.J. Atallah, and S. Macfaddin. Efficient parallel algorithms for string editing and related problems. *SIAM J. Comput.*, 19(5):968–988, 1990.
- [12] J. L. Bates and R. L. Constable. Proofs as programs. *ACM Transactions*

- on *Programming Languages and Systems*, 7(1):113–136, January 1985.
- [13] J. Bentley. *Programming Pearls*. Addison-Wesley, 1986.
 - [14] J. V. Braun and H. G. Müller. Statistical methods for DNA sequence segmentation. *Statist. Sci.*, 13:142–162, 1998.
 - [15] M. Csuros. Algorithms for finding maximal-scoring segment sets. In *Proceedings WABI2004 - 4th Workshop on Algorithms in Bioinformatics*, Lecture Notes in Computer Science. Springer Verlag, 2004.
 - [16] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. A. Kokhar. A randomized parallel 3d convex hull algorithm for coarse grained multicomputers. In *Proceedings SPAA'95 - ACM Symposium on Parallel Algorithms and Architectures*, pages 27–33. ACM Press, 1995.
 - [17] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proc. ACM 9th Annual Computational Geometry*, pages 298–307, 1993.
 - [18] L. G. Valiant et al. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 943–972. MIT Press/Elsevier, 1990.
 - [19] Y. X. Fu and R. N. Curnow. Maximum likelihood estimation of multiple change points. *Biometrika*, 77:563–573, 1990.
 - [20] J.W. Hunt and T. Szymansky. An algorithm for differential file comparison. *Comm. ACM*, (20):350–353, 1977.
 - [21] S. Karlin and V. Brendel. Chance and significance in protein and dna sequence analysis. *Science*, 257:39–49, 1992.
 - [22] R. J. Klein, Z. Misulovin, and S. R. Eddy. Noncoding RNA genes identified in AT-rich hyperthermophiles. *Proc. Natl. Acad. Sci. USA*, 99(11):7542–7547, 2002.
 - [23] W. Li, P. Bernaola-Galván, F. Haghghi, and I. Grosse. Applications of recursive segmentation to the analysis of DNA sequences. *Comput. Chem.*, 26:491–510, 2002.
 - [24] M. Lu. Parallel computation of longest-common-subsequences. volume 468 of *Lecture Notes in Computer Science*, pages 385–394. Springer Verlag, 1990.
 - [25] M. Lu and H. Lin. Parallel algorithms for the longest common subsequence problem. *IEEE Transaction on Parallel and Distributed Systems*, 5(8):835–848, 1994.
 - [26] K. Perumalla and N. Deo. Parallel algorithms for maximum subsequence and maximum subarray. *Parallel Processing Letters*, 5(3):367–373, 1995.
 - [27] P. A. Pevzner. *Computational Molecular Biology - An Algorithmic Approach*. The MIT Press, 2000.

- [28] K. Qiu and S. G. Akl. Parallel maximum sum algorithms on interconnection networks. Technical report, Queen's University, Department of Computer and Information Science, 1999. No. 99-431.
- [29] C. Rick. New algorithms for the longest common subsequence problem. Technical report, Institut für Informatik, Universität Bonn, 1994.
- [30] W. L. Ruzzo and M. Tompa. A linear time algorithm for finding all maximal scoring subsequences. In *Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology*, pages 234–241. AAAI Press, August 1999.
- [31] E. L. G. Saukas and S. W. Song. A note on parallel selection on coarse grained multicomputers. *Algorithmica*, 24:371–380, 1999.
- [32] J. Schmidt. All highest scoring paths in weighted graphs and their application to finding all approximate repeats in strings. *SIAM J. Computing*, 27(4):972–992, 1998.
- [33] A. Tiskin. All semi-local longest common subsequences in subquadratic time. In *Proceedings International Computer Science Symposium in Russia*, volume 3967 of *Lecture Notes in Computer Science*, pages 352–363. Springer Verlag, 2006.
- [34] L. Valiant. A bridging model for parallel computation. *Communication of the ACM*, 33(8):103–111, 1990.
- [35] Zhaofang Wen. Fast parallel algorithm for the maximum sum problem. *Parallel Computing*, 21:461–466, 1995.