Algoritmos paralelos de granularidade grossa para problemas de alinhamento de cadeias

Carlos Eduardo Rodrigues Alves

TESE APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO GRAU DE DOUTOR
EM
CIÊNCIA DA COMPUTAÇÃO

Orientador: **Prof. Dr. Siang Wun Song**

— São Paulo, Dezembro de 2002 —

Algoritmos paralelos de granularidade grossa para problemas de alinhamento de cadeias

Este exemplar corresponde à redação final
da tese devidamente corrigida e defendida
por Carlos Eduardo Rodrigues Alves e
aprovada pela comissão julgadora.

São Paulo, 6 de Dezembro de 2002.

Banca Examinadora:

- Prof. Dr. Valmir Carneiro Barbosa
- Prof. Dr. Nei Yoshihiro Soma
- Prof. Dr. José Augusto Ramos Soares
- Prof. Dr. Edson Norberto Cáceres
- Prof. Dr. Siang Wun Song

à minha família

Agradecimentos

Ao meu orientador, Prof. Dr. Siang Wun Song, pela dedicação e confiança em mim depositadas.

Ao Prof. Dr. Edson Norberto Cáceres, que juntamente com o prof. Song colaborou com conselhos, *brainstorms* e pacientes sugestões de texto.

Ao Dr. Frank Dehne, que com o Prof. Cáceres e o Prof. Song deu início a muito do que é apresentado neste trabalho.

Aos professores do Instituto de Matemática e Estatística da USP, pelas dicas, motivação e "puxadas de orelha" ministradas em aula e pausas para o café. Não citarei nomes para não ser injusto com nenhum por esquecimento.

Ao Prof. Dr. Nei Yoshihiro Soma, do Instituto Tecnológico de Aeronáutica, que me reencaminhou para onde minha vocação deveria me levar, e ao Prof. Venâncio Barbieri, que me encorajou a prosseguir nesta rota.

Aos meus colegas professores da Universidade São Judas Tadeu, em especial ao Prof. Dr. Angelo Sebastião Zanini e ao Prof. Ulisses Ribeiro da Silva Neto, pelo apoio e compreensão quando precisei me dedicar a este trabalho.

Aos membros da banca de avaliação desta tese que ainda não foram citados, pelas diversas correções sugeridas: Prof. Dr. José Augusto Ramos Soares (meu professor em nada menos do que três disciplinas), Prof. Dr. Valmir C. Barbosa e Prof. Dr. Nalvo F. de Almeida Jr. (suplente).

À Sandra, que me acompanhou ao longo da etapa final deste trabalho, sem dúvida a mais intensa. A paciência é uma de suas virtudes...

Resumo

Problemas de alinhamento de cadeias são fundamentais em aplicações diversas, das quais se destacam as relacionadas à Biologia Molecular. O alinhamento de duas cadeias A e B indica quão semelhantes elas são ou quais operações são necessárias para transformar uma em outra.

Uma variação do problema de alinhamento de cadeias envolve comparar a cadeia A com todas as subcadeias de B. Para este problema, são conhecidos algoritmos seqüenciais que o resolvem em tempo $O(|A||B|\log(|A|+|B|))$, um dos quais é apresentado nesta tese. É também apresentado um algoritmo seqüencial de tempo $O(|A||B|)$ para um caso especial de alinhamento de todas as subcadeias, que envolve a busca pela maior subseqüência comum a duas cadeias.

Propomos novos algoritmos paralelos para estes problemas, utilizando um modelo próprio para máquinas de memória distribuída, o CGM (*Coarse Grained Multicomputers*). Um dos objetivos fundamentais no desenvolvimento de algoritmos CGM é a redução do número de rodadas de comunicação, se possível tornando-o dependente apenas do número de processadores (p). Os algoritmos aqui propostos apresentam aceleração (*speed-up*) linear e apenas $O(\log p)$ etapas de comunicação. Não há algoritmos do nosso conhecimento com tais características na literatura.

Abstract

String alignment problems are essential to several applications, particularly to some related to molecular biology. The alignment between two strings A and B indicates how similar they are or what operations are necessary to transform one into the other.

A variation of the string alignment problem involves the comparison of a string A with all the substrings of B. For this problem, algorithms are known that have time complexity $O(|A||B|\log(|A|+|B|))$, one of which is presented in this thesis. We also present an algorithm that has time complexity $O(|A||B|)$ for a special case of alignment problem, that involves the search for the longest common subsequence of two strings.

For these problems, we propose new parallel algorithms, under a model suitable to distributed memory systems, the CGM (*Coarse Grained Multicomputers*). One of the goals in the development of algorithms under the CGM model is to attain a small number of communication rounds, if possible one that depends only on p, the number of processors. The proposed algorithms present linear speed-ups and only $O(\log p)$ comunication rounds. To the best of our knowledge, there is no algorithm for these problems with the suggested properties in the literature.

Sumário

1 Introdução e Definições Preliminares **1**

1.1 Cadeias, Subcadeias e Subseqüências 2

1.2 Problemas de Comparação de Cadeias 2

1.2.1 Edição de Cadeias 3

1.2.2 Alinhamento de Cadeias 4

1.2.3 Maior Subseqüência Comum 5

1.3 Alinhamento de Todas as Subcadeias 6

1.3.1 Aplicações do Problema ATS 6

1.3.2 Comparação Entre o Problema ATS e o de Alinhamento 8

1.4 Modelos de Computação Paralela de Granularidade Grossa 9

1.4.1 Modelo BSP (*Bulk Synchronous Parallel*) 10

1.4.2 Modelo LogP 11

1.4.3 Modelo CGM (*Coarse Grained Multicomputer*) 11

1.4.4 Comentários 12

1.5 Descrição do Trabalho 13

2 Algoritmos Fundamentais **14**

2.1 Programação Dinâmica para Problemas de Alinhamento de Cadeias 14

2.1.1 Procedimento Básico 14

2.1.2 Obtenção do Alinhamento Usando Espaço Linear 16

2.1.3 Algoritmo Ingênuo para o Problema ATS 18

2.2 Algoritmo Paralelo Básico para Alinhamento de Cadeias 18

2.3 Matrizes Monotônicas e Totalmente Monotônicas 21

2.3.1 Problema dos Mínimos das Colunas de uma Matriz Monotônica 22
2.3.2 Problema dos Mínimos das Colunas de uma Matriz Totalmente Monotônica 25
2.3.3 Contração de Linhas em Matrizes Totalmente Monotônicas 29
2.4 Comentários 31

3 Problema do Alinhamento de Todas as Subcadeias 32
3.1 Definições Preliminares 33
3.2 Divisão-e-Conquista para o ATS 35
3.2.1 Matrizes Monotônicas no Problema ATS 37
3.2.2 Algoritmo de União de GDAGs 39
3.3 Algoritmo Seqüencial para o ATS 43
3.4 Algoritmo Paralelo para o ATS 44
3.4.1 Distribuição das Matrizes AL 45
3.4.2 Divisão do Problema da União de GDAGs 46
3.4.3 Determinação dos Subproblemas 47
3.4.4 Previsão de Custos de um Subproblema 49
3.4.5 Escalonamento dos Subproblemas aos Processadores 51
3.4.6 Distribuição e Resolução dos Subproblemas 53
3.4.7 Resumo e Análise do Processo Paralelo de União de GDAGs 54
3.4.8 Análise do Algoritmo CGM para o ATS 55
3.5 Comentários 56

4 Problema da Maior Subseqüência Comum 57
4.1 Propriedades do Problema ALCS 58
4.2 Algoritmo Seqüencial para o ALCS 63
4.2.1 Propriedades do ALCS Usadas no Algoritmo Seqüencial 64
4.2.2 Descrição e Análise do Algoritmo Seqüencial 66
4.3 Estrutura Compacta para Armazenamento de D_G 69
4.4 Idéia Básica do Algoritmo CGM para o Problema ALCS 75
4.5 União de Soluções Parciais 76
4.5.1 Princípio Básico do Processo de União de Soluções 76

4.5.2 Eliminação de Redundâncias entre Subproblemas 81
4.5.3 Determinação dos Mínimos das Colunas dos Blocos Comuns 83
4.5.4 Representação das Matrizes $ContMD[i]$ 83
4.5.5 Determinação das linhas i_0 a $i_0 + r$ de D_U 84
4.5.6 Análise Completa do Processo de União 85
4.6 Análise do Algoritmo CGM para o Problema ALCS 86
4.7 Obtenção da Maior Subseqüência Comum 88
4.7.1 Procedimento Básico 88
4.7.2 Determinação dos Vértices Intermediários 89
4.7.3 Análise do Processo de Obtenção da Maior Subseqüência Comum 92

5 Conclusões **94**

A Cálculos de Complexidade **96**

Lista de Figuras

1.1 Exemplos de alinhamento 5

1.2 Exemplo de solução para o problema da Maior Subseqüência Comum 5

1.3 Alinhamento entre A e F_k, quando as similaridades entre C e todas as subcadeias de A são conhecidas 7

2.1 GDAG para o Problema de Alinhamento 16

2.2 Divisão-e-Conquista para determinação do melhor alinhamento entre A e B . . . 17

2.3 Processamento dos blocos de um GDAG durante o alinhamento paralelo de cadeias 19

2.4 Determinação dos mínimos de colunas em matrizes monotônicas 22

2.5 Operação de redução de matriz 27

3.1 GDAG G e sua matriz AL_G 34

3.2 União de subgrafos com $u = 2$ 35

3.3 Matrizes AL_S, AL_I e AL_U 36

3.4 Dados necessários para o cálculo de um bloco de AL'_U 47

4.1 GDAG para o problema ALCS 59

4.2 Demonstração da Propriedade 4.1(3) 60

4.3 Penúltimo vértice de cada caminho em um GDAG 66

4.4 Armazenamento dos dados de D^i_G em DR_G 70

4.5 Inclusão de D^{i+1}_G em DR_G a partir dos dados referentes a D^i_G 72

4.6 Unidade de alocação em DR_G 72

4.7 União de soluções parciais do ALCS 75

4.8 Construção de $Diag[W, M, l_0]$ 78

4.9 Demonstração de monotonicidade de submatriz 2×2 de $MD[i]$ quando todos os componentes são finitos 80

4.10 Estrutura das matrizes $MD[i_0]$ e $MD[i_0 + r]$, destacando r blocos comuns 82

4.11 Obtenção do caminho entre $T_U(i)$ e $F_U(j)$ 89

4.12 Obtenção do vértice intermediário do caminho entre $T_U(i)$ e $F_U(j)$ 93

Lista de Tabelas

1.1 Comparação entre os tempos de pior caso necessários para resolver problemas de Alinhamento e LCS seqüencialmente . . . 8

4.1 C_G referente ao GDAG da Figura 4.1 . . . 62

4.2 D_G referente ao GDAG da Figura 4.1 . . . 62

4.3 V_G referente ao GDAG da Figura 4.1 . . . 62

4.4 Resultados parciais da execução do Algoritmo 4.1 para o GDAG da Figura 4.1 . 69

4.5 $MD[2]$ considerando um GDAG formado pela união de dois GDAGs iguais ao da Figura 4.1 . . . 78

4.6 C_G referente ao GDAG da Figura 4.1(reprise) . . . 91

4.7 $D_{G^{rev}}$ referente ao GDAG da Figura 4.1 . . . 91

4.8 V_G referente ao GDAG da Figura 4.1(reprise) . . . 91

Convenções de Notação

Toda notação empregada é explicada ao longo do próprio texto. Esta seção explica parte da convenção utilizada, servindo como referência durante a leitura.

Cadeias e Subcadeias

As cadeias de caracteres (*strings*) são denotadas por letras maiúsculas, como A e B. Itens relacionados a determinada cadeia (comprimento, caracteres isolados etc.) envolvem o nome da própria cadeia. Caracteres isolados de uma cadeia são denotados pela letra minúscula correspondente, com índices subscritos que variam de 1 ao comprimento da cadeia. O comprimento da cadeia A é denotado por n_a ou $|A|$, assim temos:

$$A = a_1 a_2 a_3 \dots a_{n_a - 1} a_{n_a}$$

Uma subcadeia (*substring*) é indicada pela letra maiúscula correspondente à cadeia original e índices que indicam a posição inicial (subscrito) e final (sobrescrito) da subcadeia. Por exemplo

$$A_3^6 = a_3 a_4 a_5 a_6$$

Matrizes e Submatrizes

Matrizes (ou vetores) serão identificadas por nomes iniciados por letras maiúsculas. Subscritos são utilizados para diferenciar matrizes de uma mesma "classe". Quando uma seqüência de matrizes de mesmo nome é usada, cada matriz é identificada por um índice entre *colchetes*. Matrizes de qualquer dimensão terão seus componentes especificados através de índices entre *parênteses*. Para simplificar explicações ao longo do texto, os valores iniciais destes índices (relativos ao primeiro componente, ou ao componente do canto superior esquerdo) podem ser 0 ou 1, conforme o caso. Assim, um vetor unidimensional V de tamanho n pode ser indexado por valores entre 0 e $n-1$ ou entre 1 e n. Quando uma matriz é definida, os valores iniciais dos seus índices são especificados.

Quando for necessário especificar uma submatriz de uma matriz, será dado o nome da matriz usada como base, índices para as linhas e colunas e as propriedades a serem satisfeitas por estes índices para que uma linha ou coluna esteja na submatriz. As propriedades serão dadas entre colchetes. Por exemplo, a submatriz de uma matriz $n \times m$ M contendo apenas as linhas pares e a segunda metade das colunas é assim denotada:

$$M(2i, j)[1 \leq i \leq n/2, \lfloor m/2 \rfloor \leq j \leq m]$$

Uma única linha de índice i de uma matriz M pode ser indicada como um vetor através do uso de um índice sobrescrito (M^i). Elementos individuais desta linha podem ser indicados por um índice adicional ($M^i(j) = M(i, j)$).

Notação de Iverson

A notação de Iverson [21] é usada em alguns momentos do texto, sempre precedida por um comentário explicativo. Nesta notação, uma afirmação (verdadeira ou falsa) escrita entre colchetes é considerada uma expressão numérica. Se a afirmação for verdadeira, o valor da expressão é 1, caso contrário é 0. Por exemplo, $[n > 0]$ vale 1 caso n seja positivo, 0 caso contrário.

Esta notação é bastante útil na manipulação de somatórias. Sempre que ela é aplicada isto é claramente apontado no texto.

Capítulo 1

Introdução e Definições Preliminares

O recente interesse no mapeamento de genomas e as aplicações que decorrem da disponibilidade de grandes bases de dados de biologia molecular evidenciaram a necessidade de algoritmos eficientes para manipulação de cadeias de símbolos. Afinal, o genoma de qualquer espécie é formado por cadeias construídas a partir de quatro símbolos básicos: A, C, G e T (ou U), representando as 4 bases nitrogenadas que formam o Ácido Desoxirribonucleico (DNA) ou Ácido Ribonucleico (RNA). Há também muito interesse no mapeamento de proteínas, formadas por cadeias de aminoácidos (20 tipos diferentes).

Em particular, problemas de comparação de cadeias de símbolos são importantes para identificação de similaridades entre genes de diferentes espécies, identificação de novos genes ou mutações, etc.

Uma forma de aumentar a velocidade destas comparações é aplicar processamento paralelo. Em particular, é interessante procurar soluções que possam ser aplicadas em sistemas largamente disponíveis, como *clusters* de computadores com memória distribuída.

Neste trabalho serão abordados alguns problemas envolvendo comparações de cadeias de símbolos. Estes problemas surgem em diversas aplicações além da comparação de cadeias de DNA ou proteínas. Também surgem em outras aplicações, como correção e comparação de textos.

A seguir definimos alguns conceitos fundamentais sobre cadeias, explicando a notação usada neste trabalho. Nas seções seguintes definimos alguns problemas básicos envolvendo comparações de cadeias, enfatizando os que são abordados nos próximos capítulos. Também será apresentado o modelo CGM de computação paralela (*Coarse Grained Multicomputer*), utilizado ao longo deste trabalho.

1.1 Cadeias, Subcadeias e Subseqüências

Uma cadeia de símbolos (ou *string*) é uma seqüência finita de símbolos tomados de um determinado alfabeto finito. O alfabeto é dependente da aplicação e o seu tamanho pode influir na complexidade dos algoritmos. Exemplos típicos de alfabetos para aplicação em biologia computacional são o conjunto das 4 bases nitrogenadas do Ácido Desoxirribonucleico (A, C, T e G) e o conjunto de 20 aminoácidos envolvidos na construção de proteínas.

As cadeias serão denotadas por letras maiúsculas, como A e B. O comprimento da cadeia A será denotado por n_a ou $|A|$. Caracteres isolados de uma cadeia serão denotados pela letra minúscula correspondente, com índices subscritos que variam de 1 ao comprimento da cadeia. Assim temos:

$$A = a_1 a_2 a_3 \ldots a_{n_a - 1} a_{n_a}$$

A *concatenação* de duas cadeias A e B, denotada AB, é a cadeia de comprimento $n_a + n_b$ formada pelos símbolos de A seguidos pelos símbolos de B, ou seja

$$AB = a_1 a_2 \ldots a_{n_a} b_1 b_2 \ldots b_{n_b}$$

Sendo A, B e C três cadeias tais que $C = AB$, dizemos que A é um *prefixo* de C e B é um *sufixo* de C.

Uma *subcadeia* (ou *substring*) de uma cadeia A é um prefixo de algum sufixo de A (ou sufixo de algum prefixo de A). Pode-se dizer que uma subcadeia de A é uma cadeia equivalente a um "trecho" de símbolos contíguos de A.

Uma subcadeia será indicada pela letra maiúscula correspondente à cadeia original e índices que indicam a posição inicial (subscrito) e final (sobrescrito) da subcadeia. Por exemplo

$$A_3^6 = a_3 a_4 a_5 a_6$$

Usando os índices 1 e n_a podemos indicar prefixos e sufixos de A por esta mesma notação.

Uma *subseqüência* de uma cadeia A é uma cadeia de símbolos selecionados de A que preserva a ordem em que estes estavam em A. Alternativamente, uma subseqüência de A é obtida pela *remoção* de 0 ou mais símbolos de A, mantendo os demais na ordem original. Note que os símbolos presentes numa subseqüência não precisam estar contíguos na cadeia original, ao contrário do que ocorre na definição de subcadeias.

1.2 Problemas de Comparação de Cadeias

Há várias maneiras de classificar problemas de comparação de cadeias. Um critério importante nesta classificação é o número de cadeias envolvidas, que pode ser tão baixo quanto 2 (ou mesmo

1, como em [40], quando se procura similaridades entre trechos distintos de uma mesma cadeia) e tão alto quanto se queira. Neste trabalho, iremos abordar comparações entre duas cadeias.

De maneira geral, na comparação entre duas cadeias tentamos verificar o quão similares elas são, encontrando coincidências entre trechos ou verificando quais são as modificações necessárias em uma das cadeias para que possamos obter a outra. Esta última forma de comparação é genericamente denominada *edição de cadeia* (*String Editing*), sendo amplamente utilizada e podendo ser adaptada para diversas circunstâncias.

1.2.1 Edição de Cadeias

Nos problemas de edição de cadeias, várias operações de edição podem ser consideradas, cada qual com um custo associado. O problema de comparação entre A e B passa ser a busca por uma seqüência de operações de edição que transforme A em B e apresente custo total mínimo. Este custo total é denominado *distância de edição.*

Várias operações são consideradas na literatura, dentre as quais podemos citar

- Inserções e remoções de símbolos. Usualmente as penalizações para inserção e remoção são iguais. Inserções ou remoções envolvendo vários símbolos contíguos (toda uma subcadeia) podem ter um penalização diferente do total para inserção/remoção dos símbolos isolados, dependendo da aplicação.
- Substituição de símbolos. A substituição pode ter uma penalização uniforme para todos os pares de símbolos diferentes ou pode ser diferente para cada par de símbolos. Um valor positivo ou nulo pode ser dado para a "substituição" de um símbolo por ele mesmo.
- Remoção de prefixo e/ou sufixo. Em alguns casos busca-se o melhor casamento entre uma cadeia A e qualquer subcadeia de B, caso em que a remoção de prefixo ou sufixo de B tem penalização nula.
- Inversão, repetição ou translocação de subcadeia. Por inversão entende-se a remoção de todos os símbolos de uma subcadeia e sua reinserção na mesma posição, mas com os símbolos na ordem inversa. Por repetição entende-se a inserção de uma subcadeia de A em A, de forma que a nova cadeia passa a conter duas cópias da mesma subcadeia. A translocação é a remoção de uma subcadeia e sua reinserção em outro ponto da cadeia original.

Destas operações, as últimas (inversão, repetição e translocação) não serão abordadas neste trabalho. Embora elas sejam incomuns em muitas aplicações, elas têm seu lugar na biologia molecular, na explicação de certas mutações cromossômicas [22, capítulo 8]. As demais operações são de uso mais comum.

A escolha das operações de edição que serão consideradas, assim como dos custos associados a cada uma delas, depende da aplicação, da origem das cadeias e o que se entende por "similaridade" entre elas. Por exemplo, consideremos uma aplicação em que se deseja verificar

a competência de um digitador, comparando um texto fonte com o resultado de sua digitação. Inserções ou remoções de símbolos isolados são comuns durante a digitação, assim como substituições. A penalização de substituições pode ser dependente da proximidade das letras no teclado. Remoções/inserções de palavras completas podem ter um custo específico, assim como inversões de dois símbolos consecutivos.

Certas medidas de distância de edição são clássicas. Por exemplo, se estamos interessados apenas nas operações de substituição (custo 1 para símbolos diferentes, 0 para símbolos iguais) a distância de edição é denominada *distância de Hamming*. Incluindo neste esquema as operações de inserção e remoção, ambas de custo 1, temos a chamada *distância de Levenshtein* [30]. Como exemplo, tomemos as cadeias ACTTCAT e ATTCACG: a distância de Hamming é 5, devido às diferenças entre os símbolos nas posições 2, 4, 5, 6 e 7. Por outro lado, a distância de Levenshtein é 3, pois pode-se obter a segunda cadeia a partir da primeira, removendo-se o segundo símbolo (C), substituindo-se o último T por C e inserindo um G no final.

O caso em que apenas operações de remoção e inserção são permitidas também é clássico, sendo comentado na Seção 1.2.3.

1.2.2 Alinhamento de Cadeias

Neste trabalho, consideraremos operações de substituição, remoção e inserção de símbolos isolados, com custos genéricos. Com tais operações, os problemas de comparação podem ser visualizados como problemas de *alinhamento* de cadeias. Um alinhamento entre as cadeias A e B é um arranjo de duas linhas, sendo a primeira preenchida com símbolos de A e a segunda com símbolos de B. Os símbolos são colocados no arranjo preservando-se a ordem na cadeia original, com eventuais espaços (aqui representados pelo símbolo “ – ”) colocados entre eles, de forma que a remoção dos espaços em uma linha leva à cadeia original. Os espaços são inseridos de tal forma que não exista uma coluna com dois espaços no arranjo. Cada espaço na primeira linha representa uma operação de inserção, e cada espaço na segunda linha representa uma remoção.

A cada coluna do alinhamento é atribuído um valor, com base nos símbolos presentes e em um certo *esquema de atribuição de valores* (o equivalente à atribuição de custos às operações de edição). Este esquema pode ser definido por uma função σ: sendo x e y dois símbolos do alfabeto em uso, incluindo “ – ” neste alfabeto, o valor de uma coluna que contém x e y é $\sigma(x, y)$. O valor de um alinhamento é dado pela soma dos valores de suas colunas.

Os esquemas de atribuição de valores que serão aqui considerados dão valores maiores para colunas com símbolos iguais e valores menores para as colunas com símbolos diferentes, ou que representem operações de inserção ou remoção. Procurando o alinhamento de valor *máximo* teremos uma medida do quanto as cadeias são *similares*. Isto é análogo a procurar uma seqüência de operações de edição que apresente custo mínimo.

A seguir temos uma representação de dois alinhamentos possíveis para as cadeias ACTTCAT e ATTCACG (Figura 1.1). No esquema de atribuição de valores usado neste exemplo, colunas com símbolos coincidentes têm valor 1, outras colunas têm valor 0.

Podemos então definir o problema do Alinhamento de Cadeias:

A	A	C	T	T	C	A	–	T	
B	A	T	T	C	–	A	C	G	
valor	1	0	1	0	0	1	0	0	3

A	A	C	T	T	C	A	–	T	
B	A	–	T	T	C	A	C	G	
valor	1	0	1	1	1	1	0	0	5

Figura 1.1: Exemplos de alinhamento.

Definição 1.1 (Problema do Alinhamento de Cadeias) *Dadas as cadeias A e B, encontrar o alinhamento entre elas que apresente valor máximo.*

O valor do alinhamento máximo é chamado também de *similaridade* entre as duas cadeias. Em muitas situações, não estamos interessados no alinhamento em si, mas apenas na similaridade. Algoritmos baseados em programação dinâmica, apresentados no Capítulo 2, podem determinar esta similaridade em tempo $O(n_a n_b)$ e espaço $O(\min\{n_a, n_b\})$. A obtenção do alinhamento propriamente dito pode envolver maiores requisitos de tempo e espaço, mas a complexidade assintótica se mantém quadrática para o tempo e linear para o espaço, como será visto no Capítulo 2.

Os conceitos de *similaridade* e *distância de edição* são muito semelhantes quando se considera apenas substituições, remoções e inserções de símbolos isolados. De fato, uma vez conhecidos os custos de operação de edição, os valores de alinhamentos de símbolos e os comprimentos das cadeias, pode-se obter a distância de edição a partir da similaridade e vice-versa. Em [41] é apresentada uma comparação detalhada entre estes dois conceitos.

Neste trabalho, serão usados os conceitos de *alinhamento de cadeias* e *similaridade*.

1.2.3 Maior Subseqüência Comum

Um caso particular do Problema de Alinhamento de Cadeias é o Problema da Maior Subseqüência Comum, ou LCS (do inglês *Longest Common Subsequence*):

Definição 1.2 (Problema da Maior Subseqüência Comum – LCS) *Dadas as cadeias A e B, encontrar a maior cadeia C que é subseqüência de A e B.*

O problema LCS é um problema de Alinhamento de Cadeias em que apenas operações de inserção e remoção são admitidas, ou seja, não são permitidas colunas com símbolos não coincidentes a menos que um dos símbolos seja um espaço. Colunas com espaços têm valor 0 e colunas com símbolos coincidentes têm valor 1. Na Figura 1.2 temos a representação da solução do problema LCS para as cadeias já usadas nos exemplos anteriores.

A	A	C	T	T	C	A	–	–	T	
B	A	–	T	T	C	A	C	G	–	
valor	1	0	1	1	1	1	0	0	0	5

Figura 1.2: Exemplo de solução para o problema da Maior Subseqüência Comum. No caso, a subseqüência seria ATTCA.

Novamente, em muitas situações não há interesse em encontrar a maior subseqüência comum, apenas o seu comprimento, por ser uma medida de similaridade entre as cadeias.

O fato de substituições não serem admitidas no problema LCS permite que técnicas diferenciadas sejam usadas neste problema, obtendo-se algoritmos de complexidade mais baixa do que no caso de alinhamentos gerais [39]. No entanto, na análise de pior caso destes algoritmos ainda temos complexidade de tempo quadrática.

Há também algoritmos que, numa análise assintótica de pior caso, apresentam complexidade subquadrática [34, 35]. No entanto, estes algoritmos só superam os de uso geral quando as cadeias são muito grandes, maiores do que as que ocorrem em aplicações reais [23].

1.3 Alinhamento de Todas as Subcadeias

Uma extensão do Problema do Alinhamento entre Cadeias é o do Alinhamento de Todas as Subcadeias, assim definido:

Definição 1.3 (Problema do Alinhamento de Todas as Subcadeias - ATS) *Dadas A e B, duas cadeias de comprimentos n_a e n_b respectivamente, encontrar o alinhamento entre A e todas as possíveis subcadeias B_i^j de B, $1 \leq i \leq j \leq n_b$.*

Como há $O(n_b^2)$ subcadeias de B, pode-se esperar uma maior complexidade de tempo e espaço na sua solução do que no caso do alinhamento simples entre A e B. Porém, para algumas versões restritas do problema ATS existem algoritmos mais eficientes, como será visto nesta tese.

Há uma série de variações possíveis para o que se considera uma solução para o problema ATS. De maneira geral, espera-se encontrar uma estrutura de dados que permita consultas sobre o *valor* do alinhamento de A com uma determinada subcadeia B_i^j em tempo reduzido, se possível $O(1)$. A determinação do alinhamento propriamente dito leva tempo $\Omega(\max\{n_a, j-i\})$, devido ao próprio comprimento do alinhamento. Dependendo da aplicação, pode-se tolerar um aumento nestes tempos de consulta desde que o tempo ou o espaço requeridos para construção da estrutura sejam reduzidos.

1.3.1 Aplicações do Problema ATS

O problema ATS, com possíveis variações, surge como parte de problemas maiores. Alguns destes problemas são citados a seguir.

Alinhamento de uma cadeia com várias outras que possuem subcadeia comum

Em certas situações, das quais a busca em bases de dados biológicos é um dos exemplos mais evidentes, uma *cadeia-alvo* A deve ser comparada com várias outras *cadeias-fonte* $F_1, F_2, \ldots, F_n$, sendo que cada uma destas cadeias-fonte possui uma subcadeia comum C, ou seja, $F_k = P_k C S_k$,

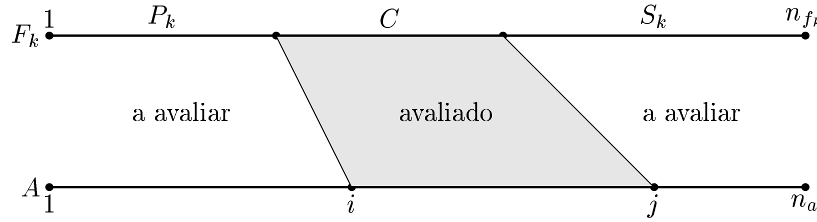

Figura 1.3: Alinhamento entre A e F_k, quando as similaridades entre C e todas as subcadeias de A são conhecidas.

$1 \leq k \leq n$. Considera-se que as cadeias $F_1, F_2, \ldots, F_n$ apresentam semelhanças por terem uma "origem" comum (por exemplo, podem ser cadeias de DNA com um ancestral comum) e foram pré-processadas para determinação da cadeia comum C.

Em [29], é apresentada uma forma de evitar usar a subcadeia comum C em todos os alinhamentos entre A e as cadeias-fonte. Para isto, determina-se primeiramente a similaridade entre C e todas as subcadeias de A. As similaridades entre A e uma certa cadeia-fonte F_k podem então ser determinadas mais rapidamente, analisando-se as similaridades entre P_k e vários prefixos de A e entre S_k e vários sufixos de A (ver Figura 1.3). Os detalhes são apresentados em [29].

Alinhamento Circular

O Problema do Alinhamento Circular [33, 40] consiste em determinar o melhor alinhamento entre uma cadeia A e qualquer *deslocamento cíclico* de uma cadeia B, ou seja, determinar o valor de i que forneça o melhor alinhamento entre A e $b_i b_{i+1} \ldots b_{n_b} b_1 b_2 \ldots b_{i-1}$.

Este problema pode ser aplicado em comparações que envolvam cadeias circulares de ácidos nuclêicos, como plasmídeos e cromossomos bacterianos ou de organelas (como mitocôndrias e cloroplastos) [22].

Este problema pode ser resolvido buscando-se os alinhamentos entre A e todas as subcadeias de BB (ou seja, a cadeia B concatenada consigo mesma) que tiverem comprimento igual a n_b.

Determinação de Repetições Aproximadas Concatenadas

A determinação de repetições concatenadas em uma cadeia A envolve buscar alinhamentos entre subcadeias concatenadas A_i^k e A_{k+1}^j. Este problema é abordado em [40] e resolvido através de uma variação do problema ATS: alinhamento de todos os sufixos de uma cadeia A com todos os prefixos de uma cadeia B. As técnicas envolvidas na solução deste problema e do ATS são análogas. De fato, a técnica de solução para o ATS apresentada no Capítulo 3 desta tese resolve ambos os problemas simultaneamente.

1.3.2 Comparação Entre o Problema ATS e o de Alinhamento

Como já comentado, o problema ATS envolve resultados mais abrangentes, o que pode levar a uma maior complexidade de tempo e espaço na sua resolução. De fato, os algoritmos conhecidos para o alinhamento simples e para o ATS diferem nas complexidades de tempo por um fator logarítmico (ver Tabela 1.1).

No entanto, quando consideramos o problema LCS, a sua extensão para um problema que envolva todas as subcadeias (que chamaremos ALCS, de *All-substrings Longest Common Subsequence*) pode ser resolvida com a mesma complexidade de tempo que o LCS básico. O Problema ALCS é definido a seguir.

Definição 1.4 (Maior Subseqüência Comum-Todas as Subcadeias (ALCS)) *Dadas as cadeias A e B, encontrar a maior subseqüência comum entre A e todas as possíveis subcadeias B_i^j de B, $1 \leq i \leq j \leq n_b$.*

A Tabela 1.1 sumariza os resultados conhecidos até o momento para os problemas de Alinhamento e LCS e suas extensões para problemas que envolvem todas as subcadeias. Estes resultados serão explicados ao longo desta tese. Deve-se notar que esta tabela apresenta a complexidade de tempo dos algoritmos de uso *prático* conhecidos no momento. Como já mencionado, para o problema LCS há algoritmos assintoticamente melhores, mas utilizáveis apenas quando as cadeias envolvidas são excepcionalmente grandes [23, 34, 35].

$A \times B$	$A \times$ Subcadeias de B
Alinhamento: $O(n_a n_b)$	ATS: $O(n_a n_b \min\{n_a, n_b\})$
LCS: $O(n_a n_b)$	ALCS: $O(n_a n_b)$

Tabela 1.1: Comparação entre os tempos de pior caso necessários para resolver problemas de Alinhamento e LCS seqüencialmente. Os algoritmos considerados são os de maior uso na prática.

Há também algoritmos que apresentam melhor complexidade de caso médio, quando aplicados em situações particulares. Por exemplo, Wu et al. [44] apresentam um algoritmo adequado para quando se espera que a maior subseqüência comum seja longa, enquanto Apostolico e Guerra [8] apresentam um algoritmo apropriado para o caso oposto. Em [39] é apresentado um algoritmo adequado para ambas as situações, mas este algoritmo, assim como os anteriores, ainda apresenta uma complexidade essencialmente quadrática no pior caso.

Temos então que a resolução do problema ALCS dá, em tempo adequado, uma solução também para o problema LCS. O aumento de complexidade se restringe a uma (pequena) constante multiplicativa.

Além das aplicações comentadas para o problema ATS, há uma outra vantagem em pesquisar este tipo de problema. A maior quantidade de informações fornecidas na resposta permite paralelizações do tipo divisão-e-conquista, como veremos ao longo desta tese. A união de soluções de subproblemas para formar a solução de um problema maior é possível graças às informações adicionais. No caso do problema LCS, a abordagem do problema ALCS torna possível a criação de um algoritmo paralelo adequado também para o LCS.

1.4 Modelos de Computação Paralela de Granularidade Grossa

O estudo dos algoritmos paralelos apresenta uma dificuldade que em geral não acomete o estudo dos algoritmos seqüenciais: a grande variedade de arquiteturas paralelas torna difícil a definição de um modelo de computação que seja universalmente aceitável. Algoritmos desenvolvidos para um certo modelo podem não ser diretamente mapeáveis em certas arquiteturas, pelo menos não sem uma grande perda de desempenho.

Um exemplo típico deste problema envolve o modelo PRAM (*Parallel Random-Access Machine*), talvez o mais popular dos modelos paralelos, que se baseia na existência de vários processadores ligados a uma única memória compartilhada. Algoritmos desenvolvidos para este modelo podem ser difíceis de aplicar em máquinas que apresentam memória distribuída e dependem de envio explícito de mensagens entre os processadores, através de uma rede de comunicação. Por outro lado, a definição de um modelo específico para máquinas de memória distribuída eventualmente teria que levar em consideração a topologia da rede de comunicação, a taxa de transferência de dados, custos de roteamento pela rede, etc. Algoritmos desenvolvidos tendo em vista uma arquitetura muito específica podem ser difíceis de adequar para outras arquiteturas.

Uma das funções de um modelo de computação é apresentar um padrão para o desenvolvimento unificado de soluções de problemas, para que estas soluções possam ser eventualmente adaptadas para arquiteturas reais com mínimo esforço e máximo desempenho. Ao observarmos boa parte das arquiteturas paralelas atualmente existentes, vemos que o custo de comunicação entre processadores tende a se tornar um gargalo para o desempenho (ver, por exemplo, [13]), o que justifica o desenvolvimento de algoritmos que visem minimizar estas comunicações, baseando-se principalmente em computações locais realizadas por processadores que possuem, cada um, uma memória local de tamanho razoável. Isto justifica o uso de modelos de computação paralela de *granularidade grossa*, ou seja, modelos que levam em consideração os custos de transferência de dados entre processadores.

Modelos como o CGM (*Coarse Grained Multicomputer*), que serão explicados a seguir, apresentam a vantagem de permitir que algoritmos paralelos sejam desenvolvidos sem levar em consideração os detalhes de redes de comunicação. Basicamente, tenta-se apenas minimizar o número de comunicações realizadas. O sacrifício que se faz ao ignorar detalhes das arquiteturas de computação, perdendo-se oportunidades de ganho de desempenho numa "sintonia fina" prévia entre algoritmo e arquitetura, é compensado pela maior generalidade e pela possibilidade de adequação posterior do algoritmo a cada arquitetura. Além disso, modelos simples como o CGM também permitem estudos teóricos importantes, como a determinação de limites inferiores para tempos de execução e comunicações.

Cada modelo apresenta os seus próprios preceitos de desenvolvimento de algoritmos, no que diz respeito ao desempenho a ser obtido. Quando se usa o modelo PRAM, um objetivo freqüente é obter algoritmos que levem tempo polilogarítmico ($O(\log^k n)$, onde n é o tamanho da entrada e k uma constante positiva qualquer) usando um número polinomial de processadores ($O(n^t)$, onde t é uma constante positiva qualquer). Para aplicações práticas, normalmente se espera que o número de processadores p seja bem menor do que o tamanho da entrada n, e que o tempo total de execução paralela seja $O(S(n)/p)$, onde $S(n)$ é o tempo do melhor algoritmo seqüencial

para o problema, com entrada de tamanho n. Este critério de desempenho (aceleração, ou *Speed Up*, linear) representa o bom aproveitamento de cada processador na solução do problema, pelo menos quando p é "muito menor" do que n (segundo algum critério bem definido).

O uso de modelos paralelos de granularidade grossa tem ganho grande atenção no meio acadêmico, por aliarem simplicidade no desenvolvimento de algoritmos e facilidade de adequação destes a arquiteturas reais, com resultados próximos aos previstos em teoria [12, 15, 36].

A seguir, veremos alguns dos modelos de granularidade grossa mais utilizados: BSP, LogP e CGM.

1.4.1 Modelo BSP (*Bulk Synchronous Parallel*)

O modelo BSP foi introduzido por Valiant [42, 18]. Um computador BSP é formado por três componentes:

- Um conjunto de módulos formados por processadores e memórias locais.
- Uma rede de comunicação que envie mensagens ponto a ponto entre os módulos.
- Um sincronizador que possibilite operações de *barreira*.

Uma computação BSP é dividida em *superpassos* separados por sincronizações em barreira. Em cada superpasso, um processador pode realizar computação local e receber/enviar dados de/para outros processadores. Os dados enviados em um superpasso só podem ser usados por outros processadores no superpasso seguinte.

Os seguintes parâmetros são usados para avaliar o desempenho de um computador BSP:

- n - tamanho da instância do problema a ser resolvido.
- p - número de processadores. Cada processador é indexado por um número entre 1 e p.
- L - número mínimo de ciclos entre cada sincronização em barreira (definindo a duração mínima de um superpasso).
- g - razão entre a capacidade de processamento (número de operações internas de computação por unidade de tempo, considerando todos os processadores) e capacidade de comunicação (número de mensagens de tamanho unitário que podem ser entregues pela rede, por unidade de tempo).

Através destes parâmetros é possível estimar, para um determinado algoritmo, qual o tempo total de execução, dividido em tempo de processamento e tempo de comunicação.

Na definição básica do modelo BSP [42], também denominado EREW BSP (ou XPRAM [43], ou ainda EREW-*phase* PRAM [19]), numa etapa de comunicação as mensagens não podem ser

duplicadas, combinadas ou sofrer qualquer tipo de *broadcast*. Uma variação para esse modelo é o denominado CREW BSP (ou CREW-*phase* PRAM [19]), em que uma mesma mensagem pode gerar um *broadcast* para processadores arbitrários.

Variações mais sofisticadas (permitindo combinações de mensagens, etc.) não serão consideradas aqui, por serem menos realistas.

1.4.2 Modelo LogP

O modelo LogP [13] é semelhante ao BSP, mas não requer sincronizações em barreira. Isto permite melhor sintonia entre algoritmo e máquina, sobrepondo etapas de comunicação e processamento interno em diferentes processadores. Os parâmetros do modelo LogP (que, por sinal, dão o nome ao modelo) são:

- L - um limite superior para a *latência* de comunicação entre dois processadores (o tempo entre o envio de uma mensagem curta e o seu recebimento).
- o - a penalização ("*overhead*") de um processador pelo envio de uma mensagem, correspondendo ao tempo de inatividade de um processador que está enviando uma mensagem curta.
- g - o intervalo ("*gap*") entre o envio ou recepção de dois dados. O inverso desta medida fornece a capacidade de transmissão da rede por processador (quantidade de dados enviados em um certo intervalo de tempo).
- P - o número de processadores.

Este modelo é provavelmente o modelo mais realístico dentre os três aqui apresentados, tendo sido proposto justamente para cobrir alguns pontos falhos do modelo BSP (de forma geral associados à sincronização em barreira). Ele segue na direção oposta à do modelo CGM, apresentado a seguir, buscando uma melhor representação das máquinas reais. No entanto, este modelo não parece adequado ao desenvolvimento de algoritmos gerais, uma vez que a otimização de um certo procedimento pode estar fortemente associada aos valores específicos dos parâmetros considerados.

1.4.3 Modelo CGM (*Coarse Grained Multicomputer*)

O modelo CGM foi proposto por Dehne et al. [14], servindo como uma simplificação do modelo BSP. Este modelo apresenta menos parâmetros do que o BSP, o que o torna um pouco mais difícil de ajustar a máquinas reais mas mais atraente para estudo de algoritmos gerais.

Para avaliar o desempenho de um computador CGM, os parâmetros são apenas n (tamanho da instância a ser resolvida) e p (número de processadores). Os parâmetros relativos à velocidade da rede e duração mínima de um superpasso foram descartados, em função de uma mudança de objetivos e a imposição de algumas restrições adicionais: cada processador deve ter capacidade

de memória $O(n/p)$ e em cada etapa de comunicação os processadores devem receber/enviar no máximo $h = O(n/p)$ mensagens de tamanho unitário de/para outros processadores. O processamento é dividido em rodadas de computação local e rodadas de comunicação, que se alternam.

O objetivo do desenvolvimento de algoritmos eficientes deixa de ser a simples procura por um tempo total de execução mínimo para incluir uma nova meta: reduzir o *número* de rodadas de comunicação, se possível tornando este número constante ou dependente apenas de p.

Em geral, supõe-se que n é muito maior do que p, sendo $n > p^2$ uma restrição comum [14]. Apesar destas restrições, o modelo CGM parece ser o mais adequado para o desenvolvimento preliminar de algoritmos paralelos de granularidade grossa, para posterior avaliação e sintonia fina com modelos mais realísticos como os BSP e o LogP. Na verdade, o modelo CGM tem se mostrado bastante adequado para o desenvolvimento de algoritmos paralelos escaláveis, que apresentam desempenho próximo ao previsto. Estas características têm atraído a atenção de um número crescente de pesquisadores [15].

1.4.4 Comentários

O objetivo deste trabalho é apresentar alguns algoritmos paralelos para alinhamento de cadeias, que possam ser adequados facilmente para arquiteturas paralelas distribuídas. Temos então a questão de escolher qual dos modelos é mais adequado para esta tarefa.

Apesar de sua menor precisão, o modelo escolhido foi o CGM. Os parâmetros dos demais modelos são adequados para situações em que uma determinada máquina será usada e seus parâmetros já são conhecidos. Quando os parâmetros são desconhecidos, uma série de decisões conflitantes podem surgir. Por exemplo, se a latência de comunicação (ou o tamanho do superpasso) for muito grande e a capacidade da rede de comunicação também, favorece-se o uso de poucas mensagens entre os processadores, que reúnam todos os dados necessários para a interação. No caso inverso (rede de baixa latência e baixa capacidade), mensagens fragmentadas podem ser mais vantajosas se permitirem, através da comunicação de resultados parciais entre os processadores, uma redução do total de dados a serem comunicados.

Os objetivos do desenvolvimento de um algoritmo CGM são claros. Contudo, questões como a citada anteriormente podem afetar a adaptação de um algoritmo a uma máquina específica. Além disso, alguns critérios do modelo CGM parecem ser desnecessariamente restritivos. Assim sendo, no uso do modelo CGM teremos em mente os seguintes critérios:

- A limitação do tamanho da memória local a $O(n/p)$, onde n é o tamanho da instância do problema a ser resolvido e p o número de processadores, é restritiva demais. Este critério faz com que os problemas que requerem espaço maior do que $O(n)$ não possam ser resolvidos mesmo quando $p = 1$ (seqüencial). Acreditamos que seja mais adequado ao "espírito" do modelo que o tamanho da memória local seja $O(M/p)$, onde M é o espaço requerido para a solução seqüencial do problema. Esta restrição impõe que a solução paralela não envolva um aumento no espaço total requerido pela solução seqüencial.

- Em alguns casos, a restrição anterior será ignorada. Um algoritmo CGM com uso de memória acima dos limites usuais pode ser considerado de qualidade inferior, mas a sua exposição pode ser útil. Assim, o tamanho da memória local será considerado um *fator de qualidade* mais do que um *fator restritivo.*

- Seguindo a alteração da restrição de memória, cada rodada de comunicação poderá envolver o envio/recebimento de $O(M/p)$ dados por rodada, mais exatamente o equivalente ao tamanho da memória local. Esta restrição será mantida. Quando é necessário o envio de mensagens maiores (por exemplo, se um processador tiver que realizar um *broadcast* de toda a sua memória), isto pode ser realizado em várias rodadas de comunicação.

- O foco do desenvolvimento se manterá na obtenção de aceleração linear e baixo número de rodadas de comunicação, mas os tamanhos das rodadas e das memórias locais serão também, na medida do possível, reduzidos.

1.5 Descrição do Trabalho

Esta tese propõe algoritmos paralelos CGM para o problema ATS e para o caso especial ALCS, ambos com aceleração linear no número de processadores p e $O(\log p)$ rodadas de comunicação. Em ambos os casos, o foco se concentra na obtenção das similaridades, ou seja, dos *valores* dos alinhamentos.

Como já comentado, a solução do problema ALCS é adequada também para o problema LCS. Para o caso específico em que uma solução para o LCS é procurada, o algoritmo proposto para o ALCS encontra também a subseqüência comum mais longa (não só o seu comprimento).

Não há algoritmos com tais características na literatura, além de [2], publicado como parte do desenvolvimento deste trabalho, que resume os resultados para o problema ATS. No momento em que esta tese está sendo escrita, os resultados para o problema ALCS se encontram em avaliação para publicação.

O Capítulo 2 apresenta resultados já presentes na literatura que serão úteis no restante do texto. Primeiramente são apresentados alguns algoritmos básicos para resolução de problemas de alinhamento (simples) entre duas cadeias. Estes algoritmos empregam programação dinâmica e a exposição deles irá facilitar a compreensão dos algoritmos aqui propostos. A paralelização dos algoritmos de programação dinâmica também será apresentada, para fins de comparação. Estes algoritmos paralelos apresentam $O(p)$ rodadas de comunicação.

Ainda no Capítulo 2 será apresentado o conceito de matrizes monotônicas e totalmente monotônicas [1], muito importantes para os capítulos seguintes.

O Capítulo 3 apresenta o algoritmo CGM para o problema ATS e o Capítulo 4 apresenta o algoritmo para o ALCS. Embora o segundo problema seja uma restrição do primeiro, as técnicas empregadas são completamente distintas. Parte da análise do algoritmo para o problema ATS foi colocada no Apêndice A para facilitar a leitura desta tese.

O Capítulo 5 apresenta conclusões e comentários sobre o trabalho.

Capítulo 2

Algoritmos Fundamentais

Neste capítulo, alguns resultados prévios relevantes a esta tese são apresentados. Inicialmente será visto como o problema de Alinhamento de Cadeias é resolvido através de programação dinâmica. A paralelização destes algoritmos é apresentada a seguir. A última seção deste capítulo é dedicada às matrizes monotônicas e totalmente monotônicas.

Quase todos os resultados deste capítulo já são conhecidos. A presença destes resultados aqui torna esta tese auto-contida e facilita a exposição dos seus resultados. Alguns detalhes apresentados são inéditos e precisam ser destacados pela importância que têm no restante do trabalho.

2.1 Programação Dinâmica para Problemas de Alinhamento de Cadeias

Uma forma bem conhecida de resolver problemas de alinhamento de cadeias é o uso de programação dinâmica [23, 37, 41]. Na verdade, o processo foi desenvolvido simultaneamente e de forma independente por diversos autores. Um problema de alinhamento é resolvido através de uma sucessão de subproblemas, cada um relacionado a um prefixo de A e um prefixo de B. Através dos valores dos alinhamentos de certos prefixos é possível calcular rapidamente o valor do alinhamento para prefixos maiores.

2.1.1 Procedimento Básico

Vamos considerar que o valor do alinhamento dos símbolos x e y é dado por $\sigma(x,y)$, lembrando que a função σ pode ter "–" como argumento. A função σ é tal que $\sigma(x,y) = \sigma(y,x)$, $\sigma(x,y) \leq 0$ se $x \neq y$ e $\sigma(x,y) \geq 0$ se $x = y$. No caso do problema LCS, por exemplo, teríamos $\sigma(x,y) = 1$ para $x = y$, 0 caso contrário. Colunas contendo dois símbolos "–" não serão permitidas, uma vez que estendem desnecessariamente o alinhamento.

Consideremos o alinhamento dos prefixos A_1^i e B_1^j, cujo valor será denominado $val(i,j)$

($val(0,0) = 0$). Construindo o arranjo do alinhamento para estes prefixos, o valor deste alinhamento será igual à soma dos valores de todas as colunas (ver Figura 1.1) até a penúltima, somado ao valor da última coluna, que deve conter uma das seguintes possibilidades:

- a_i e b_j, caso em que $val(i,j) = val(i-1,j-1) + \sigma(a_i,b_j)$.
- a_i e "–", caso em que b_j ocorre em uma coluna anterior e $val(i,j) = val(i-1,j) + \sigma(a_i,-)$.
- "–" e b_j, caso em que $val(i,j) = val(i,j-1) + \sigma(-,b_j)$, por razões semelhantes às do item anterior.

Este procedimento pode ser visualizado por um *Grafo Direcionado Acíclico do tipo Grade*, ou GDAG, em que os vértices estão dispostos em um arranjo $(n_a+1) \times (n_b+1)$. Numerando linhas e colunas a partir de 0, o vértice da linha i e coluna j, $G(i,j)$, representa $val(i,j)$. Os arcos são assim definidos:

- (arcos diagonais) para $1 \leq i \leq n_a$ e $1 \leq j \leq n_b$ há um arco de $G(i-1,j-1)$ para $G(i,j)$, de peso $\sigma(a_i,b_j)$.
- (arcos verticais) para $0 \leq i \leq n_a$ e $1 \leq j \leq n_b$ há um arco de $G(i,j-1)$ para $G(i,j)$, de peso $\sigma(a_i,-)$.
- (arcos horizontais) para $1 \leq i \leq n_a$ e $0 \leq j \leq n_b$ há um arco de $G(i-1,j)$ para $G(i,j)$, de peso $\sigma(-,b_j)$.

Este GDAG é representado na Figura 2.1. Para encontrar o melhor alinhamento entre A e B é preciso determinar o caminho de maior valor (maior soma dos pesos dos arcos percorridos) entre $G(0,0)$ e $G(n_a,n_b)$. Esta determinação pode ser feita facilmente através do seguinte algoritmo:

Algoritmo 2.1: **Alinhamento de Duas Cadeias por Programação Dinâmica.**
Entrada: Cadeias A e B.
Saída: Valor do Melhor Alinhamento entre A e B.

1 Para $i \leftarrow 0$ até n_a faça
1.1 $val(i,0) \leftarrow i \cdot \sigma(a_i,-)$
2 Para $j \leftarrow 1$ até n_b faça
2.1 $val(0,j) \leftarrow j \cdot \sigma(-,b_j)$
2.2 Para $i \leftarrow 1$ até n_a faça
2.2.1 $val(i,j) \leftarrow \max \begin{cases} val(i,j-1) + \sigma(-,b_j) \\ val(i-1,j-1) + \sigma(a_i,b_j) \\ val(i-1,j) + \sigma(a_i,-) \end{cases}$

fim do algoritmo.

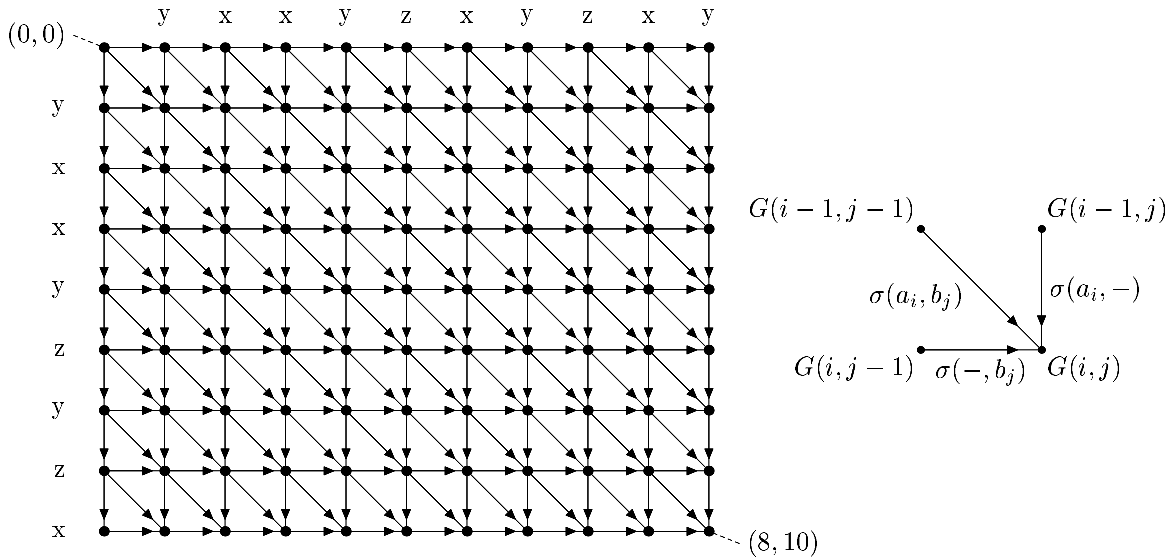

Figura 2.1: GDAG para o Problema de Alinhamento entre A = yxxyzyzx e B = yxxyzxyzxy.

O Algoritmo 2.1 calcula apenas o valor do melhor alinhamento, mas pode ser facilmente modificado para calcular também o alinhamento propriamente dito, bastando usar os valores armazenados de $val(i,j)$. Naturalmente, $G(n_a,n_b)$ pertence ao caminho no GDAG que define o melhor alinhamento. A partir deste vértice determina-se, na ordem inversa, todos os vértices que participam deste caminho. Para cada $G(i,j)$, determina-se qual dos vértices vizinhos ($G(i,j-1)$, $G(i-1,j-1)$ ou $G(i-1,j)$) foi usado no cálculo de $val(i,j)$. Isto pode ser feito observando-se os valores associados a cada vértice, mas é mais simples registrar esta informação em "ponteiros" associados a cada vértice, já durante o procedimento de cálculo.

2.1.2 Obtenção do Alinhamento Usando Espaço Linear

É fácil notar que o Algoritmo 2.1 requer tempo $\Theta(n_a n_b)$. A determinação do valor do alinhamento poderia ser feita mantendo-se apenas a última coluna de valores $val(i,j)$ determinados e desprezando os que já foram usados, mas a determinação do *caminho* no GDAG requer a manutenção de todos os valores (ou dos "ponteiros"), o que leva a espaço $\Theta(n_a n_b)$.

No entanto, graças a uma técnica descrita por Hirschberg [26], é possível determinar este caminho usando apenas espaço linear, mantendo o tempo $O(n_a n_b)$. Trata-se de uma abordagem de divisão-e-conquista: primeiramente determina-se um ponto do caminho na coluna central do GDAG, o que é feito por duas aplicações do Algoritmo 2.1:

- Determinam-se os comprimentos dos melhores caminhos de $G(0,0)$ a todo $G(i,\lfloor n_b/2 \rfloor)$, $0 \leq i \leq n_a$. Estes comprimentos equivalem aos valores $val(i,\lfloor n_b/2 \rfloor)$. A determinação e armazenamento destes valores usa espaço $O(n_a)$.
- Revertendo os arcos do GDAG, determinam-se os comprimentos dos melhores caminhos

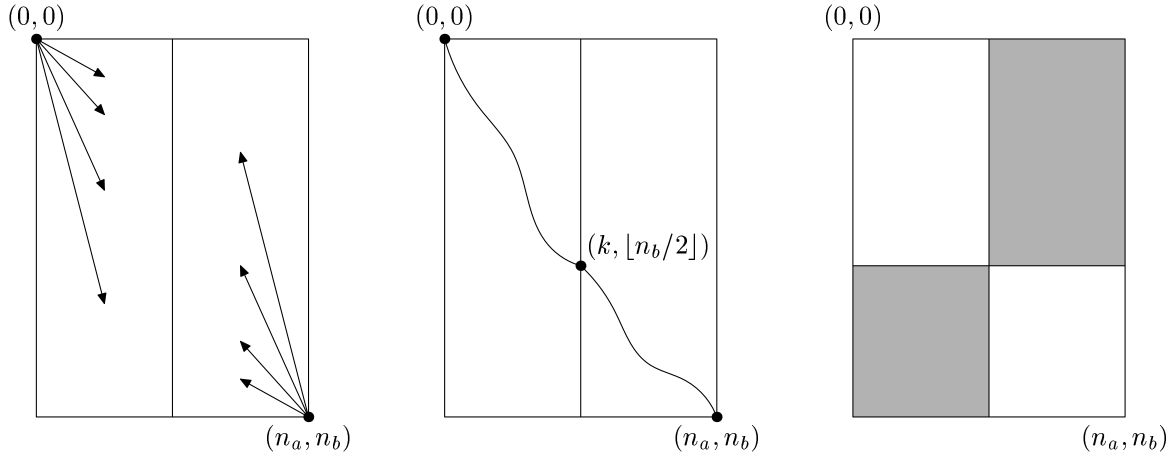

Figura 2.2: Divisão-e-Conquista para determinação do melhor alinhamento entre A e B. As áreas claras no GDAG da direita serão exploradas recursivamente.

de $G(n_a, n_b)$ a todo $G(i, \lfloor n_b/2 \rfloor)$, $0 \leq i \leq n_a$. Vamos chamar estes comprimentos de $val^r(i, \lfloor n_b/2 \rfloor)$.

Estas duas aplicações "parciais" do Algoritmo 2.1 utilizam, juntas, tempo $O(n_a n_b)$. Na verdade, o tempo total é muito próximo do tempo que seria gasto numa única execução "completa" do algoritmo (que determinasse $val(n_a n_b)$).

Determina-se então $\max_{0 \leq i \leq n_a} \{val(i, \lfloor n/2 \rfloor) + val^r(i, \lfloor n/2 \rfloor)\}$. Seja k o valor de i que corresponde a este máximo. Temos então que $G(k, \lfloor n/2 \rfloor)$ pertence ao melhor caminho entre $G(0,0)$ e $G(n_a, n_b)$. Os demais vértices podem ser determinados aplicando-se o mesmo processo recursivamente para determinar o melhor caminho entre $G(0,0)$ e $G(k, \lfloor n/2 \rfloor)$ e o melhor caminho entre $G(k, \lfloor n/2 \rfloor)$ e $G(n_a, n_b)$. A Figura 2.2 ilustra o processo.

Pela Figura 2.2 nota-se que a quantidade de valores a serem calculados nas aplicações recursivas é aproximadamente a metade da quantidade envolvida na determinação de k. Pode-se provar então que $T(n,m) = O(nm)$. De fato, $T(n,m)$ é aproximadamente o dobro do tempo gasto na determinação de k.

O espaço requerido, além do necessário para armazenamento das próprias cadeias, é $O(n_a + \log n_b)$. O termo $\log n_b$ se refere à manutenção da pilha de recursão.

Com estes resultados, demonstramos o seguinte teorema:

Teorema 2.1 *Dadas duas cadeias A e B de comprimentos n_a e n_b, respectivamente, pode-se determinar o melhor alinhamento entre A e B em tempo $O(n_a n_b)$ e espaço $O(n_a + \log n_b)$.*

Este resultado serve também, naturalmente, para o problema LCS.

2.1.3 Algoritmo Ingênuo para o Problema ATS

A partir dos resultados anteriores, deduz-se que é possível resolver o problema ATS (quando apenas as similaridades são desejadas) em tempo $O(n_a n_b^2)$, através de n_b aplicações do Algoritmo 2.1. Deve-se notar que, além da similaridade entre A e B, este algoritmo também determina as similaridades entre A e todos os prefixos de B: para A e B_1^j, este valor é $val(n_a, j)$.

Assim, para encontrar os alinhamentos envolvendo todas as subcadeias de B, basta lembrar que uma subcadeia de B é um prefixo de algum sufixo de B. Usando-se o Algoritmo 2.1 n_b vezes, uma para cada sufixo de B, temos todos os alinhamentos desejados.

Este procedimento de "força bruta" não utiliza diversas propriedades interessantes do problema ATS, que serão exploradas no Capítulo 3. Note que a abordagem aqui explicada envolve procurar, para todos os vértices na linha superior do GDAG, o melhor caminho para cada um dos vértices na linha inferior. Isto é feito tomando um vértice da linha superior por vez.

2.2 Algoritmo Paralelo Básico para Alinhamento de Cadeias

A seguir descrevemos uma solução paralela para o problema do alinhamento de cadeias, cuja idéia básica apareceu publicada em diversas formas para diversos modelos e máquinas distintos [4, 3, 16, 27].

Apesar de simples, esta abordagem apresenta desempenho satisfatório em muitas situações práticas. A simplicidade das rodadas de comunicação permite uma avaliação mais detalhada do desempenho esperado nos modelos BSP e CGM. Alguns parâmetros do algoritmo podem ser ajustados para garantir um melhor desempenho.

A idéia básica é estender o Algoritmo 2.1 fazendo com que cada processador seja responsável por uma faixa de colunas contíguas no GDAG. Dados p processadores numerados de 1 a p, o processador P_t calcula $val(i,j)$ para $0 \le i \le n_a$ e $\lfloor n_b(t-1)/p \rfloor \le j \le \lfloor n_b t/p \rfloor + 1$. Por simplicidade, vamos supor que todos os processadores ficam incumbidos de um mesmo número c de colunas.

Para evitar que o processador P_{t+1} fique ocioso enquanto o processador P_t opera, P_t envia resultados parciais para P_{t+1} em várias rodadas de comunicação. Para isto, o GDAG é também dividido em faixas horizontais de l linhas contíguas, sendo que l é um parâmetro ajustável. Assim sendo, o GDAG é dividido em blocos $l \times c$, que representam as tarefas básicas do algoritmo paralelo.

Seja $b = \lceil n_a/l \rceil$. O GDAG fica então dividido em $b \times p$ blocos, que identificaremos por $B(s,t)$, $1 \le s \le b$, $1 \le t \le p$.

A computação do bloco $B(s,t)$ é feita pelo processador P_t. A computação se desenvolve em uma "frente de onda", que começa no bloco de dados do canto superior esquerdo do GDAG e se propaga para baixo e para a direita. A Figura 2.3 ilustra este processo.

A computação do bloco $B(s,t)$ envolve três operações por parte do processador P_t:

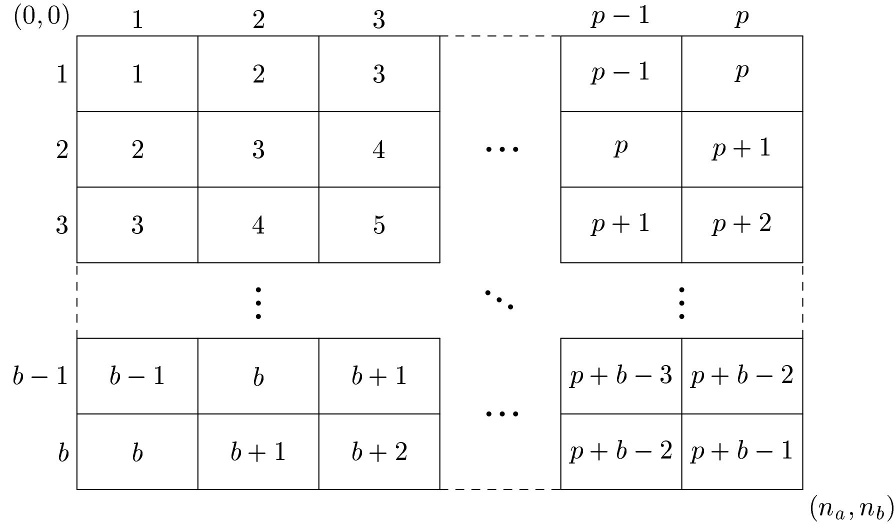

Figura 2.3: Processamento dos blocos de um GDAG durante o alinhamento paralelo de cadeias. O número no interior de cada bloco indica o superpasso em que a computação do bloco é realizada.

- Se $t > 1$, esperar pelos dados da última coluna do bloco $B(s, t-1)$. São l dados a serem enviados pelo processador P_{t-1}.
- Computar os dados de $B(s,t)$, usando os dados de $B(s, t-1)$ e $B(s-1, t)$. Estes últimos foram calculados pelo próprio processador P_t.
- Se $t < p$, enviar os l dados da última coluna de $B(s,t)$ para o processador P_{t+1}.

O número de superpassos necessários é $p + b - 1$. O parâmetro l (que define b) pode ser aumentado para reduzir o número de superpassos, mas este aumento faz com que haja mais ociosidade entre os processadores durante os primeiros e os últimos superpassos. Por outro lado, um valor muito pequeno de l faz com que o número de rodadas de comunicação se torne muito grande. A latência da rede de comunicação começa a se tornar significativa e anula o benefício da redução da ociosidade.

Estudos empíricos para o melhor valor de l são úteis no ajuste desta técnica a uma aplicação real [3, 16].

Para o modelo CGM, o caso $l = \lceil \alpha n_a / p \rceil$, onde $\alpha \leq 1$ é uma constante, fornece um algoritmo que executa em tempo $O(n_a n_b / p)$ e requer $O(p)$ rodadas de comunicação, em que cada processador transfere $O(n_a/p)$ dados [4].

Esta técnica pode ser modificada para permitir a determinação do alinhamento em si (e não apenas do seu valor) com espaço linear, usando uma variação do algoritmo de Hirshberg [16, 27]. Temos então o seguinte resultado:

Teorema 2.2 *O alinhamento de duas cadeias A e B, respectivamente de comprimentos n_a e n_b*

($n_a \leq n_b$), pode ser determinado no modelo CGM com $p \leq n_b$ processadores em tempo $O(n_a n_b/p)$ e espaço local $O(n_a+n_b)$, usando $O(p)$ rodadas de comunicação em que cada processador transfere $O(n_a/p)$ dados.

Como já comentado, este resultado é suficiente em muitas aplicações em que apenas o alinhamento entre duas cadeias é requerido. Este algoritmo pode, naturalmente, ser usado para o LCS, como um caso especial de alinhamento.

A técnica utilizada pode ser aplicada também em modelos de granularidade fina. Na verdade, a técnica aqui apresentada é um caso particular de paralelização de procedimento de programação dinâmica [17]. Os subproblemas (no caso, alinhamentos de prefixos de A com prefixos de B) apresentam dependências entre si, estabelecendo uma ordem parcial (representada pelo GDAG). Uma ordenação topológica utilizando esta ordem parcial indica em que etapa cada subproblema pode ser resolvido. No caso, os valores dos alinhamentos de prefixos A_1^i e B_1^j tais que $i + j = k$ podem ser resolvidos na etapa $k - 1$.

Há muitas outras abordagens possíveis para a paralelização de problemas de alinhamento. Quando modelos de granularidade fina são empregados, abordagens de divisão-e-conquista são comuns na obtenção de algoritmos de tempo polilogarítmico, ou seja, que executam em tempo $O(\log^k(n_a + n_b))$ para algum k. Nestas abordagens, o GDAG é dividido em blocos que são processados em paralelo, gerando resultados que são então unidos. O processamento de cada bloco envolve a divisão deste em blocos menores, recursivamente. Exemplos desta abordagem podem ser encontrados em [6, 38].

Em geral, esta abordagem apresenta um problema sério: os algoritmos paralelos obtidos não são ótimos quando comparados ao algoritmo seqüencial (o produto entre o número de processadores e o tempo de execução paralelo não é $O(n_a n_b)$). Ainda assim, o algoritmo de Apostolico et al. [6] expõe técnicas importantes para a resolução do problema ATS, como será visto no Capítulo 3.

Outra forma de explorar paralelismo em problemas de comparação de cadeias é realizar simultaneamente a comparação entre uma cadeia *alvo* e *várias* outras, provenientes de uma base de dados, para encontrar as de maior similaridade. Este é um problema de grande importância prática, mas que foge ao escopo desta tese. Os desafios neste tipo de problema se concentram mais no balanceamento de carga do que na forma de colaboração entre os processadores [45].

Nos capítulos 3 e 4 serão mostrados algoritmos CGM que determinam os valores dos alinhamentos de A e todas as subcadeias de B (problema ATS), usando apenas $O(\log p)$ rodadas de comunicação. No Capítulo 3, em que não se considera nenhuma restrição para os pesos nos arcos, o tempo de computação é maior do que o necessário para o Problema do Alinhamento (simples) por um fator logarítmico. Mas no Capítulo 4, em que o problema ALCS é abordado, o tempo de computação é (assintoticamente) o mesmo que o necessário para o problema LCS. Disto resulta um algoritmo melhor para o problema LCS, com apenas $O(\log p)$ rodadas de comunicação.

2.3 Matrizes Monotônicas e Totalmente Monotônicas

Nesta seção serão apresentados dois tipos de matrizes de grande importância no restante deste texto: as *matrizes monotônicas* e as *matrizes totalmente monotônicas*.

Estas matrizes surgem em vários tipos de aplicações. Um exemplo geométrico é o de determinar para cada vértice de um polígono convexo qual é o vértice mais distante dele. Este exemplo é apresentado por Aggarwal et al. em [1], que é a base para a apresentação que se segue. As definições e algoritmos foram adaptados aqui, contendo alguns detalhes, análises e considerações que não constavam no artigo original. Algumas destas considerações são bastante importantes para a análise de resultados desta tese.

As propriedades características das matrizes monotônicas e totalmente monotônicas dizem respeito à localização dos elementos mínimos em cada coluna (em algumas aplicações, o interesse pode estar nos elementos máximos, ou nas linhas). Antes da apresentação destas propriedades, alguns termos devem ser definidos:

Definição 2.1 *[$Imin[M]$ e $Cmin[M]$] Dada uma matriz $n \times m$ M de números reais, o vetor $Imin[M]$ é tal que para todo j, $1 \leq j \leq m$, $Imin[M](j)$ é o menor valor de i tal que $M(i,j)$ é o mínimo da coluna j de M. O vetor $Cmin[M]$ contém os valores mínimos das colunas de M, ou seja, $Cmin[M](j) = M(Imin[M](j), j)$.*

Assim, $Imin[M]$ contém as *localizações* dos mínimos das colunas, enquanto $Cmin[M]$ contém os *valores* mínimos propriamente ditos. Passamos então às definições principais desta seção:

Definição 2.2 (Matriz Monotônica) *Seja M uma matriz $n \times m$ de números reais. M é uma matriz* monotônica *se, para $1 \leq j_1 < j_2 \leq m$, $Imin[M](j_1) \leq Imin[M](j_2)$.*

Definição 2.3 (Matriz Totalmente Monotônica) *Seja M uma matriz $n \times m$ de números reais. M é uma matriz* totalmente monotônica *se toda submatriz 2×2 de M for monotônica.*

Lema 2.3 *Toda matriz totalmente monotônica é também monotônica.*

Prova. Se M não é uma matriz monotônica existem j_1 e j_2 tais que $j_1 < j_2$ e $Imin[M](j_1) > Imin[M](j_2)$. A matriz 2×2 formada por $M(i_1, j_1)$, $M(i_1, j_2)$, $M(i_2, j_1)$ e $M(i_2, j_2)$ não é monotônica, logo M não pode ser totalmente monotônica. □

Em muitas aplicações há interesse em determinar os elementos mínimos de *todas* as colunas de uma matriz monotônica ou totalmente monotônica. O problema de determinação dos mínimos das colunas de uma matriz é definido a seguir. O restante desta seção é dedicado à resolução deste problema utilizando as propriedades já comentadas.

Definição 2.4 (Problema dos Mínimos das Colunas) *Dada uma matriz $n \times m$ de números reais M, determinar o vetor $Imin[M]$.*

O problema é definido em termos de *localizar* os mínimos. A obtenção de $Cmin[M]$ a partir de $Imin[M]$ é feita de acordo com a definição.

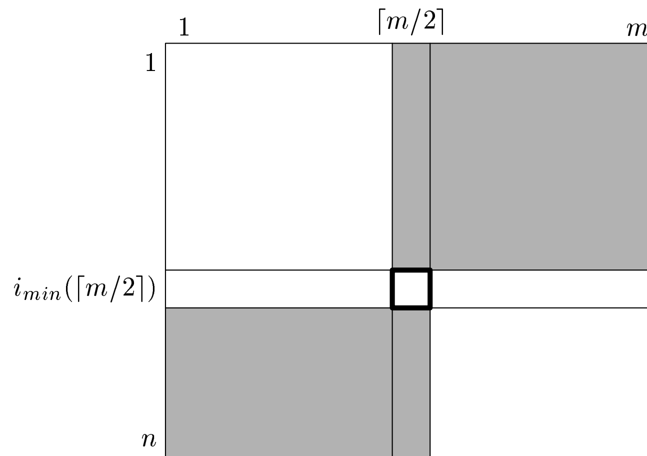

Figura 2.4: Determinação dos mínimos de colunas em matrizes monotônicas. O mínimo da coluna central foi determinado, as áreas escuras podem ser ignoradas. Nas áreas claras o procedimento será repetido recursivamente.

2.3.1 Problema dos Mínimos das Colunas de uma Matriz Monotônica

Para matrizes monotônicas, o algoritmo (recursivo) de determinação dos mínimos nas colunas é dado a seguir. O algoritmo é definido em termos de submatrizes devido à recursão.

Algoritmo 2.2: **Determinação dos mínimos das colunas de uma matriz monotônica.**
Entrada: Submatriz $M(i,j)[i_1 \leq i \leq i_2, j_1 \leq j \leq j_2]$.
Saída: $Imin[M](j)$, $j_1 \leq j \leq j_2$.

1 $aux \leftarrow i_1$
2 $meio \leftarrow \lfloor (j_1 + j_2)/2 \rfloor$
3 Para $i \leftarrow i_1 + 1$ até i_2 faça
3.1 Se $M(i, meio) < M(aux, meio)$ então
3.1.1 $aux \leftarrow i$
4 $Imin[M](meio) \leftarrow aux$
5 Se $meio > j_1$ resolva o problema para a submatriz $M(i,j)[i_1 \leq i \leq aux, j_1 \leq j < meio]$.
6 Se $meio < j_2$ resolva o problema para a submatriz $M(i,j)[aux \leq i \leq i_2, meio < j \leq j_2]$.

fim do algoritmo.

A Figura 2.4 ilustra a operação do Algoritmo 2.2.

Teorema 2.4 *Os mínimos de todas as colunas de uma matriz monotônica M $(n \times m)$ podem ser encontrados em tempo $O(n \log m + m)$ e espaço $O(m)$ (suficiente para a resposta).*

Prova. A prova é baseada no Algoritmo 2.2. Os passos 1 e 3 determinam o mínimo da coluna central da submatriz enquanto os passos 5 e 6 cuidam das colunas à esquerda e à direita da central, respectivamente. Recursivamente, todas as colunas são tratadas. Deve-se observar que os passos 5 e 6 trabalham com submatrizes das quais foram eliminadas algumas linhas. Pela monotonicidade de M, estas linhas não podem conter o mínimo das colunas a serem tratadas, o que garante que o algoritmo funciona corretamente.

Vamos avaliar o tempo do algoritmo através do número de acessos a M que ele deve fazer. Considerando uma submatriz $n \times m$, podemos provar que, sendo $t(n,m)$ o maior número possível de acessos realizado pelo algoritmo, temos $t(n,m) \leq f(n,m) = (\lfloor \log m \rfloor + 1)(n-1) + 2m - 1$. De fato, testando para pequenos valores de m temos $f(n,1) = n$, $f(n,2) = 2n+1$, $f(n,3) = 2n+3$ e $f(n,4) = 3n+4$.

Fazemos então uma indução em m. Para uma matriz $n \times m$, temos n acessos devidos aos passos 1 e 3, mais os acessos devidos às chamadas recursivas dos passos 5 e 6. Seja $x = i_{min}(\lceil m/2 \rceil)$. Como cada chamada recursiva envolverá no máximo $\lfloor m/2 \rfloor$ colunas e como $\lfloor \log \lfloor m/2 \rfloor \rfloor = \lfloor \log m \rfloor - 1$, segundo a hipótese de indução devemos ter

$$
\begin{aligned}
t(n,m) \leq\ & n + \underbrace{f(x, \lfloor m/2 \rfloor)}_{passo\ 5} + \underbrace{f(n-x+1, \lfloor m/2 \rfloor)}_{passo\ 6} = \\
& n + \underbrace{\lfloor \log \lfloor m/2 \rfloor + 1 \rfloor (x-1) + 2\lfloor m/2 \rfloor - 1}_{passo\ 5} + \underbrace{\lfloor \log \lfloor m/2 \rfloor + 1 \rfloor (n-x) + 2\lfloor m/2 \rfloor - 1}_{passo\ 6} \leq \\
& n + \lfloor \log m \rfloor (x-1) + \lfloor \log m \rfloor (n-x) + 2m - 2 = n + \lfloor \log m \rfloor (n-1) + 2m - 2 = \\
& (\lfloor \log m \rfloor + 1)(n-1) + 1 + 2m - 2 = f(n,m).
\end{aligned}
$$

Isto prova o limite superior. Como $f(n,m) = O(n \log m + m)$, concluímos a demonstração. Observe que o valor de x, que depende da posição do mínimo da coluna central, não tem influência na determinação do limite superior. □

O termo m no tempo de execução se deve, de fato, à necessidade de processar todas as colunas, mesmo quando as eliminações possíveis graças à monotonicidade da matriz fazem com que certas colunas tenham apenas um candidato a mínimo restante. Relaxando este requisito, o tempo de execução pode ser reduzido para $O(n \log m)$, o que é vantajoso para matrizes "largas" $(m \gg n)$. No entanto, os mínimos das colunas não podem ser explicitados. Em [1] é demonstrado que o limite inferior para o tempo de solução do problema dos mínimos das colunas é de fato $\Omega(n \log m)$ no caso de matrizes monotônicas, mas é indicado que o limite superior é também $O(n \log m)$, desconsiderando o problema de não tornar explícitos os mínimos.

O Algoritmo 2.2 pode ser adaptado para executar em tempo $O(n \log m)$ quando $m \gg n$, gerando informações sobre os mínimos. Infelizmente, estas informações não permitem a determinação destes mínimos em tempo constante. As informações são guardadas em um vetor J de tamanho $n+1$ (índices iniciando em 1), de tal modo que $Imin[M](j) = i_1$ para todo j,

$J(i_1) \leq j < J(i_1 + 1)$. Desta forma, dado um certo j, para se determinar $Imin[M](j)$ é preciso fazer uma busca binária no vetor J, o que toma tempo $O(\log n)$. Deve-se notar que $J(1) = 1$ e definiremos $J(n+1) = m+1$. Podemos chamar estes $J(i)$ de *pontos de início*, pois eles indicam a coluna em que os mínimos começam a aparecer na linha i (e deixam de aparecer na linha $i-1$).

A seguir temos a adaptação do Algoritmo 2.2 que determina os pontos de início.

Algoritmo 2.3: **Determinação dos mínimos das colunas de uma matriz monotônica "larga".**
Entrada: Submatriz $M(i,j)[i_1 \leq i \leq i_2, j_1 \leq j \leq j_2]$.
Saída: $J(i)$, $i_1 + 1 \leq i \leq i_2$.

1 Se $j_1 \leq j_2$ então
1.1 $aux \leftarrow i_1$
1.2 $meio \leftarrow \lfloor (j_1 + j_2)/2 \rfloor$
1.3 Para $i \leftarrow i_1 + 1$ até i_2 faça
1.3.1 Se $M(i, meio) < M(aux, meio)$ então
1.3.1.1 $aux \leftarrow i$
1.4 Se $aux > i_1$ resolva o problema para a matriz $M(i,j)[j_1 \leq j < meio, i_1 \leq i \leq aux]$.
1.5 Se $aux < i_2$ resolva o problema para a matriz $M(i,j)[meio < j \leq j_2, aux \leq i \leq i_2]$.
2 Senão
2.1 Para $i \leftarrow i_1 + 1$ até i_2 faça
2.1.1 $J(i) \leftarrow j_1$

fim do algoritmo.

Teorema 2.5 *Para uma matriz monotônica M $(n \times m)$ com $n < m$ pode-se montar uma estrutura em tempo $O(n \log m)$ e espaço $O(n + \log m)$ que permite consultas ao elemento mínimo de qualquer coluna em tempo $O(\log n)$.*

Prova. A demonstração é feita com base no Algoritmo 2.3. Ao invés de determinar o mínimo da coluna central, este algoritmo apenas usa a posição deste mínimo para dividir o problema em problemas menores, com o objetivo de determinar os pontos de início.

Para demonstrar que o algoritmo opera corretamente, estabelecemos, apenas para fins de demonstração, que $Imin[M](0) = 1$ e $Imin[M](m+1) = n$. Supomos também que a chamada inicial ao Algoritmo 2.3 é feita com a matriz M completa como argumento (ou seja, $i_1 = j_1 = 1$, $i_2 = n$ e $j_2 = m$) e que $n > 1$. No caso de $n \leq 1$, a execução do algoritmo é desnecessária.

Temos os seguintes invariantes no início de *todas* as chamadas:

1. $Imin[M](j_1 - 1) = i_1$.

2. $Imin[M](j_2 + 1) = i_2$.
3. $i_1 < i_2$

É fácil perceber que as chamadas recursivas nas linhas 1.4 e 1.5 conservam estes invariantes.

Cada chamada estabelece o valor de $J(i)$ para $i_1 < i \leq i_2$. Isto pode ser demonstrado por indução no número de colunas da submatriz. Quando este número é 0 ($j_1 > j_2$, o que no caso implica $j_1 = j_2 + 1$), o ponto de início para todas as linhas de $i_1 + 1$ a i_2 é j_1 e o laço da linha 2.1 estabelece os valores de $J(i)$. Quando há pelo menos uma coluna, a linha 1.4 estabelece $J(i)$ para $i_1 < i \leq aux$ e a linha 1.5 estabelece $J(i)$ para $aux < i \leq i_2$.

Assim, o algoritmo estabelece $J(i)$ corretamente para $1 < i \leq m$. Para finalizar, é preciso fazer $J(1) = 1$ e $J(m + 1) = n + 1$.

O espaço usado é apenas $O(n + \log m)$, necessário para o armazenamento dos pontos de início e para a pilha de recursão. Para análise do tempo vamos considerar o *nível* de cada chamada do algoritmo como sendo 0 para a chamada que envolve a matriz completa e $l + 1$ para cada chamada realizada recursivamente por uma chamada de nível l. Cada chamada envolve um certo conjunto de linhas contíguas da matriz, com pelo menos duas linhas. É fácil demonstrar que duas chamadas de um mesmo nível podem envolver, no máximo, *uma* mesma linha comum. Assim, há no máximo n chamadas em um certo nível.

Considerando todas as chamadas de um certo nível, a soma de todos os acessos feitos no laço da linha 1.3 é $O(n)$. Como o número de colunas das submatrizes é reduzido pela metade a cada nível, o número de níveis é $O(\log m)$. Com isto, concluímos que o tempo de execução do Algoritmo 2.3 é $O(n \log m)$.

Finalmente, como já comentado, para determinar $Imin[M](j)$ para um certo valor de j é preciso realizar um busca binária nos valores de $J(i)$ à procura de um i que satisfaça $J(i) \leq j < J(i + 1)$. □

2.3.2 Problema dos Mínimos das Colunas de uma Matriz Totalmente Monotônica

Para matrizes totalmente monotônicas, o processo de determinação dos mínimos nas colunas pode ser realizado em tempo linear no número de linhas, se a matriz for "estreita" ($n > m$). Graças à monotonicidade total da matriz, é possível realizar uma etapa inicial de eliminação de linhas em tempo $O(n)$, resultando em uma matriz quadrada. Esta etapa é denominada *redução*. Vamos primeiro estudar o processo de redução e depois ver como o problema dos mínimos é resolvido.

O processo de redução é baseado na comparação de dois elementos em uma mesma coluna. Com base no resultado desta comparação, pode-se determinar que certos elementos da matriz estão *mortos*, ou seja, não podem ser os mínimos de suas respectivas colunas. Mais exatamente, temos o seguinte lema:

Lema 2.6 *Seja M uma matriz $(n \times m)$ totalmente monotônica. Para $1 \leq i_1 < i_2 \leq m$ e $1 \leq j \leq m$ temos que*

1. *Se $M(i_1, j) \leq M(i_2, j)$ então todos os elementos $M(i_2, j')$ com $1 \leq j' \leq j$ estão mortos e*
2. *Se $M(i_1, j) > M(i_2, j)$ então todos os elementos $M(i_1, j')$ com $j \leq j' \leq m$ estão mortos.*

Prova. A demonstração é baseada na monotonicidade total de certas submatrizes 2×2 de M, mais exatamente as formadas pelos elementos $M(i_1, j)$, $M(i_2, j)$, $M(i_1, j')$ e $M(i_2, j')$. Se $M(i_1, j) \leq M(i_2, j)$ e $1 \leq j' \leq j$ então $M(i_1, j') \leq M(i_2, j')$ e isto prova a afirmação 1. Se $M(i_1, j) > M(i_2, j)$ e $j \leq j' \leq m$ então $M(i_1, j') > M(i_2, j')$ e fica provada a afirmação 2. □

O Algoritmo 2.4 realiza a redução usando o Lema 2.6. Este algoritmo determina linhas completas que estão mortas e as elimina de M. O resultado é a matriz R, inicialmente igual a M. A cada passo, um índice $k \geq 1$ será usado para indicar que os elementos $M(i, j)$ com $1 < i \leq k$ e $1 \leq j < i$ são considerados mortos. Se $k = 1$ não há elementos mortos em R. A Figura 2.5 ilustra este processo.

Algoritmo 2.4: Redução matriz totalmente monotônica.
Entrada: Matriz M $(n \times m)$.
Saída: Matriz R $(m \times m)$ que contém os mínimos de todas as colunas de M.

1	$R \leftarrow M$
2	$k \leftarrow 1$
3	Enquanto R tiver mais do que m linhas faça
3.1	Se $R(k, k) \leq R(k+1, k)$ então
3.1.1	Se $k < m$ então
3.1.1.1	$k \leftarrow k + 1$
3.1.2	Senão
3.1.2.1	Elimine a linha $k + 1$ de R
3.2	Senão
3.2.1	Elimine a linha k de R
3.2.2	Se $k > 1$ então
3.2.2.1	$k \leftarrow k - 1$

fim do algoritmo.

Lema 2.7 *Dada uma matriz $n \times m$ M $(n > m)$, o Algoritmo 2.4 gera, em tempo $O(n)$ e usando espaço adicional $O(n)$, uma matriz $m \times m$ R tal que os elementos mínimos de cada coluna equivalem aos elementos mínimos das colunas correspondentes de M. A solução do Problema dos Mínimos das Colunas de M é dada a partir da solução do mesmo problema para R em tempo $O(m)$.*

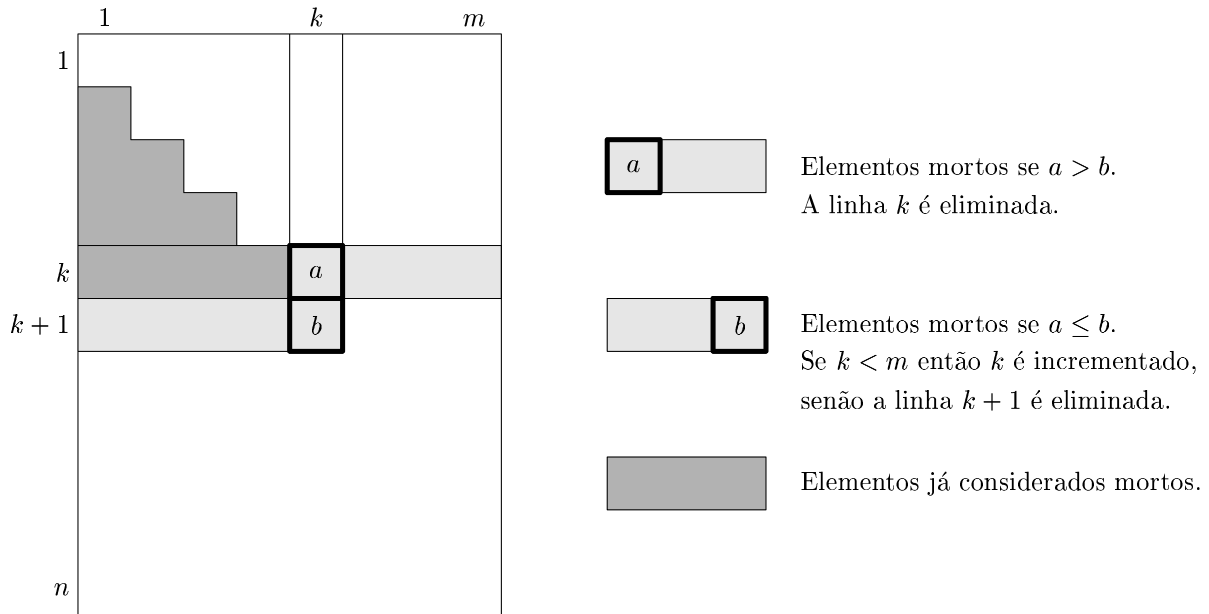

Figura 2.5: Operação de redução de matriz.

Prova. No início de cada iteração do laço da linha 3, a afirmação de que os elementos $R(i, j)$ com $1 < i \leq k$ e $1 \leq j < i$ estão mortos é invariante. De fato, com $k = 1$ nenhum elemento é considerado morto, como já foi dito, e se a afirmação é válida no início de uma iteração ela o será no final, pois

- se $R(k, k) \leq R(k+1, k)$ então pelo Lema 2.6(1) todos os elementos $R(k+1, j)$ com $1 \leq j \leq k$ estão mortos. k pode ser incrementado, mantendo o invariante, desde que isto não faça $k > m$ (caso em que toda a linha $k + 1$ estaria morta, podendo ser eliminada).
- se $R(k, k) > R(k + 1, k)$ então pelo Lema 2.6(2) os elementos $R(k, j)$ com $k \leq j \leq m$ estão mortos. Isto faz com que todos os elementos da linha k estejam mortos. A linha é eliminada e k é decrementado, a menos que isto faça $k < 1$.

A manutenção deste invariante e o fato de que as únicas linhas eliminadas são aquelas com elementos mortos pode ser facilmente percebida na Figura 2.5.

Um fato não explicitado no Algoritmo 2.4 é que a matriz R pode ser definida a partir de M com o uso de uma lista encadeada de índices de linhas, que ocupa apenas espaço $O(n)$. A atribuição da linha 1 indica a inicialização da lista encadeada em tempo $O(n)$. Cada eliminação de linha pode ser feita em tempo constante, portanto só é necessário provar que o laço da linha 3 é executado $O(n)$ vezes.

Sendo a, b e c o número de vezes que as linhas 3.1.1.1, 3.1.2.1 e 3.2.1 são executadas, respectivamente, temos que $b + c = n - m$ pois este é o número total de linhas eliminadas. Como k é iniciado com 1 e pode crescer até no máximo m, $a - c \leq m - 1$. Assim, o número de iterações do laço da linha 3 é $a + b + c \leq a + 2b + c = a - c + 2(b + c) \leq 2n - m - 1 = O(n)$.

Finalmente, a lista encadeada que define quais linhas de M estão em R pode ser copiada em um vetor, em tempo $O(m)$. Os elementos de R podem então ser acessados em tempo $O(1)$. O Problema dos Mínimos das Colunas para M pode ser resolvido com base em R. □

O Algoritmo 2.5 resolve o Problema dos Mínimos das Colunas para matrizes totalmente monotônicas utilizando o procedimento de redução já mostrado.

Algoritmo 2.5: Determinação dos mínimos das colunas de uma matriz totalmente monotônica.
Entrada: Matriz M $(n \times m)$.
Saída: $Imin[M](j)$, $1 \leq j \leq m$.

1 Aplique o Algoritmo 2.4 a M, gerando uma matriz reduzida R $(m \times m)$.

2 Se $m = 1$ então

2.1 Determine o mínimo da (única) coluna de M a partir de R e encerre.

3 Senão

3.1 $M_2 \leftarrow R(i, 2j)[1 \leq i \leq m, 1 \leq 2j \leq m]$ (M_2 tem as colunas pares de R).

3.2 Resolva recursivamente o Problema dos Mínimos das Colunas para M_2. Esta solução dá a solução para as colunas pares de R, registrada em $Imin[R](2j)$ para $1 \leq 2j \leq m$.

3.3 Determine $Imin[R](2j+1)$ para $1 \leq 2j+1 \leq m$ utilizando os resultados do passo anterior.

3.4 Determine $Imin[M](j)$ a partir de $Imin[R](j)$, para $1 \leq j \leq m$.

fim do algoritmo.

Teorema 2.8 *O Problema dos Mínimos das Colunas pode ser resolvido para uma matriz $n \times m$ totalmente monotônica em tempo $O(n + m + n\log(m/n))$ e espaço $O(n + m\log m)$.*

Prova. Vamos considerar inicialmente a operação do Algoritmo 2.5 para o caso em que $n \geq m$, depois estenderemos para o caso geral. Quando $n \geq m$, o teorema afirma que o tempo de execução é apenas $O(n)$.

Pelo Lema 2.7 o passo 1 reduz a matriz M usando tempo e espaço $O(n)$. Após a execução deste passo os resultados obtidos ocupam espaço $O(m)$, o suficiente para estabelecer a relação entre as linhas de M e R.

O passo 3.1 indica a criação de uma matriz em que apenas as colunas pares de R são consideradas. Este passo pode ser implementado criando-se um parâmetro para o algoritmo que indica a distância entre as colunas a serem usadas, a ser dobrado a cada chamada recursiva no passo 3.2. O tempo necessário é, portanto, constante.

O passo 3.3 pode ser resolvido facilmente em tempo $O(m)$. Os resultados para as colunas pares dividem as m linhas de R em $\lceil m/2 \rceil$ intervalos, um para cada mínimo a ser encontrado em uma coluna ímpar de R. A soma dos comprimentos dos intervalos é no máximo $m + \lceil (m-1)/2 \rceil =$

$O(m)$. O espaço adicional necessário para esta etapa é constante.

O passo 3.4 usa o resultado da redução no passo 1, armazenado em espaço $O(m)$, para adaptar os mínimos obtidos à matriz M.

Em resumo, O algoritmo utiliza, descontando-se a chamada recursiva no passo 3.2, tempo $O(n+m)$ e espaço $O(n+m)$. O espaço requerido para uma determinada chamada recursiva antes da próxima chamada é de apenas $O(m)$, logo o algoritmo acumula espaço $O(m \log m)$, resultando num espaço total $O(n + m \log m)$.

O tempo gasto pelo algoritmo sem considerar as chamadas recursivas é $O(n+m) = O(n)$. Na primeira chamada recursiva o tamanho da matriz é $m \times m/2$ e o tempo gasto (sem considerar as chamadas seguintes) é $O(m)$. A cada nova chamada as dimensões das matrizes são reduzidas pela metade, o que indica que o tempo total para todas as chamadas é $O(m)$. Como $n \geq m$, o tempo total gasto é $O(n)$.

O caso em que $n < m$ pode ser tratado normalmente pelo Algoritmo 2.5. Para a análise do tempo gasto, é mais simples considerar a seguinte variação: inicialmente, o Problema dos Mínimos das Colunas é resolvido para a submatriz $M(i, \lceil mj/n \rceil)[1 \leq i \leq n, 1 \leq j \leq n]$, formada por n colunas igualmente espaçadas de M. Usando o Algoritmo 2.5, esta etapa leva tempo $O(n)$ e requer espaço $O(n \log n) = O(m \log m)$.

Com a determinação dos mínimos nas n colunas selecionadas, os mínimos das demais colunas ficam restritos a n submatrizes. Cada submatriz está compreendida entre duas colunas selecionadas e entre as duas linhas em que estão os mínimos destas colunas. A aplicação do Algoritmo 2.2 a todas estas submatrizes determina os mínimos restantes da matriz em tempo $O(n \log(m/n) + m)$. O algoritmo completo leva tempo $O(n \log(m/n) + m)$ e requer espaço $O(n + m \log m)$. □

No Teorema 2.8, da mesma forma que no Teorema 2.4, o termo m no tempo de execução se deve à necessidade de explicitar os mínimos, o que leva à execução da busca pelos mínimos até em submatrizes de uma única linha. Se este termo for desprezado, o tempo resultante é $O(n \log(m/n))$. Em [1] é demonstrado que o limite inferior para o tempo necessário para encontrar os mínimos em todas as colunas de uma matriz $n \times m$ é também $\Theta(n \log(m/n))$.

Para matrizes "largas" ($m \gg n$) é interessante considerar uma nova variação do algoritmo. Utilizando o Algoritmo 2.3 no lugar do Algoritmo 2.2 (ver final da demonstração do Teorema 2.8), pode-se construir um vetor $J(i)(1 \leq i \leq n+1)$, como comentado na Seção 2.3.1, que permite consultas ao mínimo de qualquer coluna em tempo $O(\log n)$. Isto nos leva ao seguinte resultado.

Corolário 2.9 *Para uma matriz $n \times m$ totalmente monotônica M com $n < m$, pode-se montar uma estrutura em tempo $O(n \log(m/n))$ e espaço $O(m \log m)$ que permite consulta ao elemento mínimo de qualquer coluna em tempo $O(\log n)$.*

2.3.3 Contração de Linhas em Matrizes Totalmente Monotônicas

A operação de contração de linhas foi utilizada em [32], implicitamente, para resolução paralela do problema LCS no modelo CREW-PRAM. Para tornar a exposição dos resultados do

Capítulo 4 mais claras, daremos uma definição explícita para esta operação e demonstrações de alguns resultados.

Definição 2.5 *[Contração de Linhas Contíguas de uma Matriz Totalmente Monotônica] Sendo M uma matriz totalmente monotônica, uma* contração de linhas *aplicada a um conjunto de linhas* contíguas *de M é a substituição em M de todas estas linhas por uma única nova linha. O elemento da coluna i desta nova linha é o mínimo dos elementos presentes na coluna i das linhas originais substituídas.*

Assim, se $S = \{l_1, l_1{+}1, l_1{+}2, \dots, l_2\}$, denotaremos por $Cont_S(M)$ a contração de uma matriz M aplicada às linhas entre l_1 e l_2, inclusive. Seja M uma matriz $n \times m$, então $N = Cont_S(M)$ é uma matriz $(n - l_2 + l_1) \times m$ tal que $N^{l_1} = Cmin[M(i,j)[l_1 \leq i \leq l_2]]$, $N^i = M^i$ para $i < l_1$ e $N^i = M^{i+l_2-l_1}$ para $i > l_1$.

Teorema 2.10 *Se M é uma matriz totalmente monotônica e S um conjunto de índices consecutivos de linhas de M, $Cont_S(M)$ é também uma matriz totalmente monotônica e $Cmin[M] = Cmin[Cont_S(M)]$.*

Prova. $Cmin[M] = Cmin[Cont_S(M)]$ decorre diretamente da Definição 2.5. É preciso provar apenas a monotonicidade total de $Cont_S(M)$.

Seja $Cont_S(M)$ uma contração qualquer de linhas contíguas de uma matriz M totalmente monotônica. Seja uma submatriz 2×2 qualquer de $Cont_S(M)$, com elementos das colunas j_1 e j_2 ($j_1 < j_2$). Teremos apenas duas possibilidades: nenhuma das duas linhas é o resultado da contração ou uma delas o é.

No primeiro caso a submatriz é claramente monotônica, pois é uma submatriz também de M. No segundo caso teremos mais dois subcasos: os dois elementos da linha "contraída" estarão numa mesma linha em M ou não.

Novamente, no primeiro caso a submatriz será submatriz também de M. No segundo, os elementos da linha contraída virão das linhas i_1 e i_2, podendo ser escritos como $M(i_1, j_1)$ e $M(i_2, j_2)$. A monotonicidade total de M garante que $i_1 < i_2$.

Os dois outros elementos da submatriz de $Cont_S(M)$ podem ser escritos como $M(i_0, j_1)$ e $M(i_0, j_2)$. Vamos supor que $i_0 < i_1$. O caso em que $i_0 > i_2$ pode ser tratado de forma análoga.

Para provar que esta submatriz é monotônica, é preciso provar que se $M(i_1, j_1) < M(i_0, j_1)$ então $M(i_2, j_2) < M(i_0, j_2)$. A monotonicidade total de M garante que se $M(i_1, j_1) < M(i_0, j_1)$ então $M(i_1, j_2) < M(i_0, j_2)$. Como $M(i_2, j_2) < M(i_1, j_2)$ (pois $M(i_1, j_2)$ é o elemento mínimo da coluna j_2 dentro do conjunto contraído S) temos que $M(i_2, j_2) < M(i_0, j_2)$, como esperado. $\Box$

Pode-se provar também que se M é uma matriz monotônica (mas não totalmente monotônica) $Cont_S(M)$ é também monotônica.

2.4 Comentários

Neste capítulo foram apresentados alguns resultados já conhecidos, para facilitar o entendimento dos próximos capítulos e tornar este trabalho auto-contido.

Os algoritmos para alinhamento apresentados aqui são muito diferentes dos que serão apresentados nos próximos capítulos, mas os conceitos fundamentais, como a representação via GDAGs, são semelhantes.

As matrizes totalmente monotônicas serão usadas principalmente no Capítulo 4. A exposição dos resultados originalmente publicados em [1] aqui serviu a dois propósitos. O primeiro é, como já mencionado, tornar esta tese auto-contida. O segundo é tornar explícitos certos detalhes que não constavam no artigo original, em particular a análise do espaço requerido pelos algoritmos, detalhes da representação das matrizes e, principalmente, uma análise mais apurada do caso em que as matrizes são "largas".

Para as aplicações originalmente propostas para as matrizes totalmente monotônicas, o caso das matrizes "largas" era pouco relevante. Para as aplicações do Capítulo 4, este não é o caso. O Corolário 2.9 será aplicado e o fator logarítmico no tempo de acesso terá que ser levado em consideração.

Capítulo 3

Problema do Alinhamento de Todas as Subcadeias

Neste capítulo, abordaremos o problema do Alinhamento de Todas as Subcadeias (Definição 1.3). Parte dos resultados aqui presentes foram apresentados no *14th Annual ACM Symposium on Parallel Algorithms and Architectures* [2].

O problema é abordado sem que qualquer restrição seja feita com relação aos pesos das operações de edição, que se refletem nos pesos dos arcos do GDAG associado. Tudo que se requer é que estes pesos possam ser determinados em tempo $O(1)$.

Como solução para o problema ATS, espera-se encontrar uma estrutura de dados que permita consultas sobre o *valor* do alinhamento (similaridade) de A com uma determinada subcadeia B_i^j em tempo $O(1)$.

A determinação do alinhamento propriamente dito não será considerada, porque as aplicações do problema ATS normalmente não envolvem a determinação de *todos* os alinhamentos: em geral, os valores dos alinhamentos são usados em um problema maior e, em algumas circunstâncias, *um único* alinhamento deve ser obtido (o mais adequado, segundo algum critério dependente da aplicação). Manter estruturas de dados que permitam consultas a *todos* os alinhamentos é custoso e pouco útil.

Uma forma de lidar com este problema, não muito elegante, mas adequada na prática, é usar o algoritmo aqui apresentado para determinar os valores dos alinhamentos e, depois de determinar qual é o alinhamento desejado, usar o procedimento citado no Capítulo 2 para encontrar este único alinhamento. O tempo de execução do algoritmo completo ainda é determinado pela resolução do ATS. Desta forma, vamos nos concentrar unicamente na determinação dos valores dos alinhamentos.

Considerando-se o GDAG do problema do alinhamento, buscamos os melhores caminhos (ou seja, os que apresentam maior soma dos pesos dos arcos percorridos) para todos os pares de vértices em que o primeiro está na linha superior do grafo e o segundo na linha inferior.

Uma solução seqüencial óbvia para este problema, já comentada no Capítulo 2, é executar n_b+1 procedimentos de programação dinâmica como definido na Seção 2.1, cada um usando um vértice diferente da linha superior do grafo como origem e determinando os melhores caminhos para todos os vértices na linha inferior. Isto leva a um algoritmo de complexidade $O(n_a n_b^2)$.

Utilizando propriedades do GDAG, é possível resolver o problema em tempo $O(n^2 \log n)$ quando $n_a = n_b = n$. Um exemplo de algoritmo para o ATS com tal complexidade é o proposto por J. Schmidt [40], onde o problema recebe o nome de *Todos os Melhores Caminhos em Grafos Grade Rotulados*. O algoritmo apresentado constrói uma estrutura de dados a partir da qual o valor do melhor caminho entre um vértice na borda da grade e *qualquer* outro vértice (não necessariamente numa borda da grade) pode ser determinado em tempo $O(\log(n_a + n_b))$. Esta estrutura permite diversos tipos de consulta, incluindo a construção em tempo $O(n_a n_b)$ de uma matriz que permite consulta sobre qualquer caminho de borda para borda na grade em tempo $O(1)$.

Em um trabalho de Apostolico et al. [6] são estudadas soluções paralelas para o Problema de Alinhamento de Cadeias (não ATS), usando o modelo PRAM. O problema ATS é abordado como parte de uma estratégia de divisão-e-conquista, resultando em um algoritmo paralelo em que o número total de operações é $O(n^2 \log n)$. Este resultado será usado como base para os resultados deste capítulo.

Uma terceira abordagem possível envolve uma divisão do problema semelhante à de [6], mas usa técnicas da Seção 2.3.2 (matrizes totalmente monotônicas) para executar algumas etapas do procedimento. Esta abordagem é inédita (pelo nosso conhecimento) e é rapidamente comentada no final da seção 3.2.1. No entanto, ela não apresenta vantagens sobre as anteriores e não é essencial para os resultados apresentados neste capítulo.

A seguir, consideraremos o problema ATS na sua forma de encontrar caminhos no GDAG. Na verdade, abordaremos uma extensão do problema. Ao invés de determinar apenas os melhores caminhos entre os vértices do topo e os vértices do fundo do GDAG, iremos determinar o melhor caminho de cada vértice na linha superior ou na coluna mais à esquerda para cada vértice na linha inferior ou coluna mais à direita. Os caminhos da linha superior para a linha inferior dão alinhamentos entre A e todas as subcadeias de B. Envolvendo a coluna mais à esquerda e a coluna mais à direita, encontramos também os valores dos alinhamentos de subcadeias de A com B, de sufixos de A com prefixos de B e de prefixos de A com sufixos de B. Esta expansão dos resultados a serem obtidos torna mais simples a descrição do algoritmo paralelo e não afeta a sua complexidade assintótica.

3.1 Definições Preliminares

Seja G um GDAG referente a duas cadeias A e B de comprimentos n_a e n_b, como definido no Capítulo refcha:alinha. Este GDAG terá $(n_a+1)(n_b+1)$ vértices. As bordas de G, cada qual um conjunto de vértices nos limites do grafo, serão denotadas T_G (topo), F_G (fundo), E_G (esquerda) e D_G (direita).

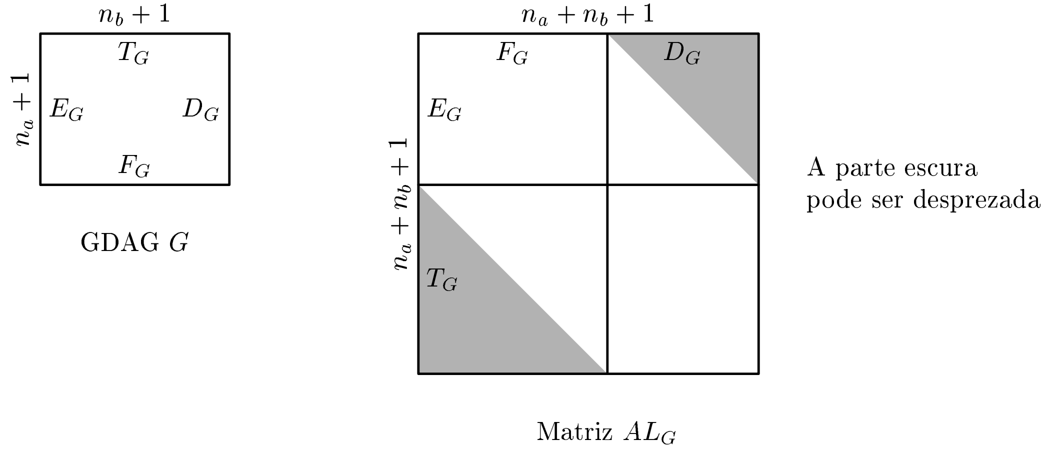

Figura 3.1: GDAG G e sua matriz AL_G.

Definimos uma seqüência $orig_G$ que contém os vértices de $E_G \cup T_G$, *origens* dos caminhos procurados. Esta seqüência é iniciada pelo vértice inferior esquerdo de G ($orig_G(0)$) e terminada pelo vértice superior direito ($orig_G(n_a+n_b)$). O vértice superior esquerdo é ($orig_G(n_a)$). Para os vértices de *destino*, pertencentes a $F_G \cup D_G$, definimos uma seqüência $dest_G$ iniciada no vértice inferior esquerdo ($dest_G(0)$), passando pelo vértice inferior direito ($dest_G(n_b)$) e terminando no vértice superior direito ($dest_G(n_a + n_b)$). Definir os índices iniciais com 0 simplifica a exposição dos algoritmos.

Os resultados do ATS (estendido como comentado anteriormente) para o GDAG G serão armazenados numa matriz de nome AL_G. A estrutura desta matriz é mostrada na Figura 3.1. $AL_G(i,j)$ terá o valor do melhor caminho entre $orig_G(i)$ e $dest_G(j)$. Note que não há caminhos entre certos pares de vértices, portanto os valores correspondentes em AL_G podem ser desconsiderados (para não deixá-los indefinidos, serão considerados $-\infty$). Note também que os índices iniciais desta matriz são 0, de acordo com as definições de $orig_G$ e $dest_G$.

Para a solução paralela, o GDAG é dividido em subgrafos e para cada um deles é resolvida uma instância menor do ATS. As soluções de cada ATS são então reunidas numa solução geral. Esta estratégia de divisão-e-conquista, a mesma que foi usada em [6], será esboçada a seguir (Seção 3.2). Além de servir para a paralelização da solução do problema, ela também é a base para o algoritmo seqüencial que pode ser usado para cada subgrafo.

Na Seção 3.4 comentaremos a solução paralela de granularidade grossa para o problema. Sendo p o número de processadores, n_a e n_b os comprimentos das cadeias ($n_a \leq n_b$), o algoritmo CGM apresentado requer tempo $O\left(\frac{n_b^2}{p} \log n_a\right)$, espaço $O\left(\frac{n_b^2}{p}\right)$ e $O(\log p)$ rodadas de comunicação, sendo que cada processador envia/recebe $O\left(\frac{n_b^2}{p}\right)$ dados a cada rodada. O número de processadores deve ser limitado a $\sqrt{n_a}$.

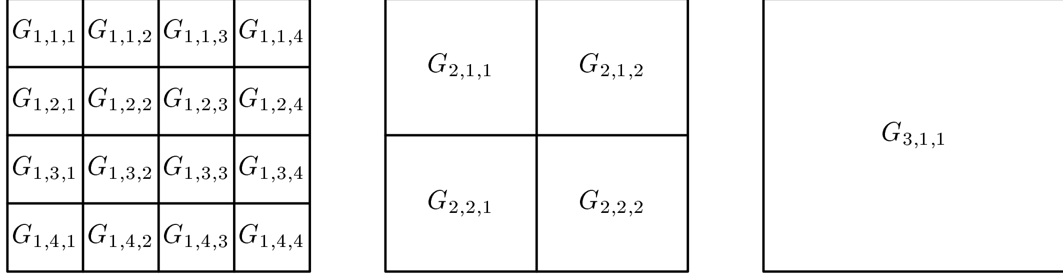

Figura 3.2: União de subgrafos com $u = 2$.

3.2 Divisão-e-Conquista para o ATS

Para simplificar a exposição dos resultados deste capítulo, será suposto, sem perda de generalidade, que o número de processadores é $p = 2^{2u}$ para algum u inteiro. Desta maneira, o GDAG pode ser dividido em $\sqrt{p}$ faixas horizontais e $\sqrt{p}$ faixas verticais, resultando em p subgrafos. Iremos supor, novamente sem perda de generalidade, que estes subgrafos são de mesmo tamanho. Dois subgrafos vizinhos apresentam uma linha ou coluna de vértices em comum, correspondendo à sua fronteira mútua.

Supondo que as matrizes de alinhamento AL correspondentes a cada um dos subgrafos já foram calculadas, o que pode ser feito por algoritmo seqüencial exposto na Seção 3.3, procedemos com a união dos subgrafos para obtenção da matriz AL_G. Os subgrafos são unidos em grupos de quatro em $u = \frac{\log p}{2}$ passos, obtendo-se subgrafos cada vez maiores, com suas respectivas matrizes AL.

Mais precisamente, denotamos cada um dos subgrafos envolvidos como $G_{r,l,c}$ onde r indica o passo do processo de união em que o subgrafo é usado, $1 \leq r \leq u+1$. Quanto maior o valor de r, maior o subgrafo. Os subscritos l e c representam a posição do subgrafo (linha e coluna) com relação aos demais subgrafos num mesmo passo do processo, $1 \leq l \leq \sqrt{p}/2^{r-1}$, $1 \leq c \leq \sqrt{p}/2^{r-1}$.

Os grafos $G_{1,l,c}$, $1 \leq l \leq \sqrt{p}$, $1 \leq c \leq \sqrt{p}$, representam os subgrafos iniciais e $G_{u+1,1,1} = G$. A partir dos subgrafos disponíveis em uma certa etapa r geramos os subgrafos da etapa seguinte $r+1$ da seguinte forma (ver Figura 3.2):

$$G_{r+1,l,c} = G_{r,2l-1,2c-1} \cup G_{r,2l-1,2c} \cup G_{r,2l,2c-1} \cup G_{r,2l,2c} \ .$$

Esta união de quatro grafos é na verdade realizada aos pares. Deve-se lembrar que o que fazemos em cada união é gerar a matriz AL do grafo resultante a partir das matrizes dos grafos utilizados.

O processo de união de dois grafos vizinhos é comentado a seguir. Consideraremos o caso particular em que os dois grafos estão um sobre o outro, possuindo uma fronteira horizontal comum. O caso dos grafos vizinhos alinhados horizontalmente é análogo. Por simplicidade, chamaremos o grafo superior de S, o inferior de I e o resultante da união de U. Suas matrizes de alinhamentos serão chamadas respectivamente de AL_S, AL_I e AL_U.

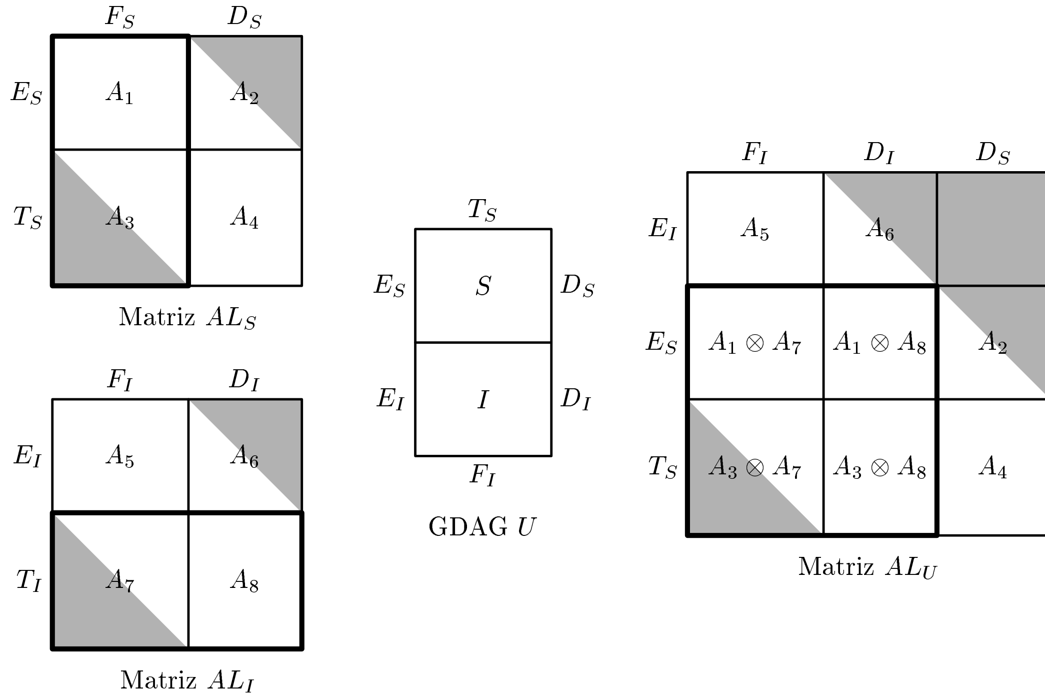

Figura 3.3: Matrizes AL_S, AL_I e AL_U. As submatrizes $AL'S$, AL'_I e AL'_U são destacadas com bordas mais grossas.

A determinação de AL_U a partir de AL_S e AL_I é esboçada na Figura 3.3. Os trechos hachurados indicam posições não utilizadas nas matrizes. Alguns valores são simplesmente copiados de AL_S e AL_I para AL_U, outros devem ser calculados. As submatrizes efetivamente usadas serão chamadas AL'_S, AL'_I e AL'_U. A Figura 3.3 ilustra estas submatrizes, destacando-as com bordas grossas.

Os elementos de AL'_U envolvem caminhos iniciados em um vértice de $orig_S$ e terminados em um vértice de $dest_I$. Todos estes caminhos devem passar por um vértice na fronteira comum $F_S = T_I$. Definimos então uma seqüência $meio_U$ com os pontos desta fronteira comum, iniciada no vértice mais à esquerda ($meio_U(0)$) e terminada no vértice mais à direita. Com grafos de dimensões iguais, os comprimentos das seqüências $orig_S$ e $dest_I$ são iguais. Chamamos este comprimento de t, $t > 1$, e o comprimento da seqüência $meio_U$ de k. Assim, as dimensões de AL'_S, AL'_I e AL'_U são, respectivamente, $t \times k$, $k \times t$ e $t \times t$.

Um algoritmo ingênuo para o cálculo de AL'_U envolve procurar, para cada um dos t^2 pares de vértices de nosso interesse, qual dos k vértices de $meio_U$ maximiza a soma dos valores dos caminhos nos grafos S e I. Para um certo par de vértices $(orig_S(i), dest_I(j))$ seria preciso utilizar toda a linha i de AL'_S e toda a coluna j de AL'_I. Este algoritmo teria complexidade $O(kt^2)$. Na verdade, trata-se de uma multiplicação de matrizes, usando as operações de *máximo* e *soma* no lugar das usuais *soma* e *multiplicação*.

Um algoritmo mais eficiente restringe as pesquisas que devem ser feitas na seqüência $meio_U$.

Definimos uma matriz M_U onde $M_U(i,j)$ é o índice do vértice *mais à esquerda* de $meio_U$ que pertence a um caminho *ótimo* entre $origs_S(i)$ e $dest_I(j)$. Caso não exista tal caminho, $M_U(i,j)$ não será utilizado, portanto não importa qual é o seu valor. A seguir temos uma definição mais precisa.

Definição 3.1 (Matriz M_U) *Sendo $0 \leq i \leq t$ e $0 \leq j \leq t$, $M_U(i,j)$ é o menor valor de l tal que $AL'_S(i,l) + AL'_I(l,j) = \min_{0 \leq v \leq k-1} \{AL'_S(i,v) + AL'_I(v,j)\}$.*

Toda vez que um valor $AL'_U(i,j)$ é calculado, o valor correspondente de $M_U(i,j)$ também é calculado e armazenado. Isto não será mencionado explicitamente daqui para a frente, a não ser quando necessário.

3.2.1 Matrizes Monotônicas no Problema ATS

As seguintes propriedades são essenciais na elaboração de um algoritmo mais eficiente, já tendo sido usadas em [6, 40].

Propriedade 3.1 *Se $i_1 < i_2$ então $M_U(i_1,j) \leq M_U(i_2,j)$, $0 \leq i_1 < i_2 < t$, $0 \leq j < t$.*

Propriedade 3.2 *Se $j_1 < j_2$ então $M_U(i,j_1) \leq M_U(i,j_2)$, $0 \leq i < t$, $0 \leq j_1 < j_2 < t$.*

Prova. Vamos abordar a Propriedade 3.1. Supondo que $i_1 < i_2$ e $M_U(i_1,j) > M_U(i_2,j)$, os caminhos ótimos de $origs_S(i_1)$ e $origs_S(i_2)$ a $dest_I(j)$ teriam que se cruzar em um vértice de S. Sendo x o ponto de cruzamento, poderíamos construir novos caminhos trocando os trechos entre x e $dest_I(j)$. Como a soma dos valores dos dois caminhos seria conservada, os novos caminhos teriam que ser ótimos. Teríamos obtido então um caminho de $origs_S(i_1)$ a $dest_I(j)$ que usa um vértice de $meio_U$ à esquerda de $meio_U(M_U(i_1,j))$, contrariando a definição de $M_U(i_1,j)$. A demonstração da Propriedade 3.2 é análoga. □

Estas propriedades sugerem a existência de certas matrizes (totalmente) monotônicas que podem ser aproveitadas no problema. Deixaremos este fato para o final da seção, pois as explicações a seguir ficariam mais complicadas se envolvessem explicitamente tais matrizes monotônicas.

Dada a Propriedade 3.1, se sabemos $M_U(i_1,j)$ e $M_U(i_2,j)$ para $i_1 < i_2$ e j, podemos determinar $M_U(i,j)$ (e portanto $AL'_U(i,j)$) para qualquer i entre i_1 e i_2 pesquisando apenas caminhos que passam entre $meio(M_U(i_1,j))$ e $meio(M_U(i_2,j))$, inclusive. Além disso, se para um certo j soubermos os valores de $M_U(i,j)$ para todo i de uma seqüência $i_0 < i_1 < \ldots < i_r$, podemos calcular $M_U(i,j)$ para um valor de i em cada intervalo na seqüência com apenas uma *varredura* de $meio_U$, em tempo $O(M_U(i_r,j) - M_U(i_1,j) + r) = O(k + r)$ (o termo r é incluído porque os r trechos de $meio_U$ considerados se sobrepõem nas bordas). Desta maneira, pode-se praticamente dobrar o número de caminhos conhecidos.

Esta operação será denominada simplesmente *varredura*. Dizemos que a varredura descrita acima é *liderada* pelo vértice $dest_I(j)$. No caso do algoritmo seqüencial, exposto na próxima

seção, estas varreduras usam $i_0 = 0$ e $i_r = t - 1$. O procedimento geral é detalhado no Algoritmo 3.1.

Algoritmo 3.1: **Varredura (liderada pelo vértice $dest_I(j)$).**
Entrada: $M_U(i,j)$ e $AL'_U(i,j)$ para um j fixo e vários valores de i numa seqüência $i_0 < i_1 < \ldots < i_r$.
Saída: $M_U(i,j)$ e $AL'_U(i,j)$ para todo i' numa seqüência $i_0 < i'_0 < i_1 < i'_1 < \ldots < i'_{r-1} < i_r$.

1 Para $q \leftarrow 0$ até $r - 1$ faça
1.1 $lmax \leftarrow M_U(i_q, j)$, $Amax \leftarrow AL_S(i'_q, lmax) + AL_I(lmax, j)$
1.2 Para $l \leftarrow M_U(i_q, j) + 1$ até $M_U(i_{q+1}, j)$ faça
1.2.1 Se $AL_S(i'_q, l) + AL_I(l, j) > Amax$ então
1.2.1.1 $lmax \leftarrow l$, $Amax \leftarrow AL_S(i'_q, l) + AL_I(l, j)$
1.3 $M_U(i'_q, j) \leftarrow lmax$, $AL'_U(i'_q, j) \leftarrow Amax$

fim do algoritmo.

Devido à Propriedade 3.2, uma varredura também pode ser feita com uma origem fixa e vários vértices de destino, ou seja, pode ser *liderada* por um vértice $origs_S(i)$. A descrição do algoritmo para este caso é análoga à descrição anterior.

Antes de passar para a descrição do algoritmo completo para união de GDAGs, uma observação a respeito dos resultados anteriores: o Algoritmo 3.1 é de fato baseado em propriedades de matrizes monotônicas, apresentadas na Seção 2.3. A partir de AL'_S e AL'_I é possível definir duas classes de matrizes totalmente monotônicas: uma para vértices de $origs_S$ e outra para vértices de $dest_I$, como descrito a seguir.

Para cada vértice $origs_S(i)$ os pesos dos possíveis caminhos até cada um dos vértices de $dest_I$ podem ser considerados como elementos de uma matriz $k \times t$ MO_i tal que $MO_i(l,j) = AL'_S(i,l) + AL'_I(l,j)$. Ao encontrar o *máximo* na coluna j desta matriz temos o valor de $AL'_U(i,j)$ e o índice de linha deste máximo indica $M_U(i,j)$. Estas considerações decorrem diretamente da Definição 3.1.

A Propriedade 3.2 indica que a matriz $-MO_i$ é monotônica, conforme a Definição 2.2 (a inversão de sinal permite que se busque elementos *mínimos* nas colunas). Na verdade, é fácil demonstrar que ela é *totalmente* monotônica, sendo que a demonstração é semelhante à já feita para as propriedades 3.1 e 3.2. De maneira semelhante, podemos definir matrizes totalmente monotônicas $-MD_j$, uma para cada vértice $dest_I(j)$.

Os procedimentos de varredura são baseados na monotonicidade das matrizes $-MO_i$ e $-MD_j$, mas não aproveitam a monotonicidade *total* destas matrizes. A aplicação de várias varreduras com um único vértice fixo, digamos $origs_S(i)$, permite o cálculo de todos os caminhos deste vértice para os de $dest_I$, mas este procedimento é menos eficiente do que o que aproveita a monotonicidade total.

Uma maneira simples de encontrar todos os melhores caminhos de vértices de $origs_S$ para vértices de $dest_I$ consiste em usar o Algoritmo 2.5 para todas as matrizes $-MO_i$, determinando os mínimos das colunas. Pelo Teorema 2.8 podemos concluir que tal procedimento levaria tempo $O(t^2 + tk\log(t/k))$, adequado se t não for muito maior do que k. Podemos obter os mesmos resultados operando-se sobre as matrizes $-MD_j$.

Este *não* será o procedimento adotado neste capítulo. A próxima seção mostra como um procedimento eficiente pode ser construído a partir das varreduras, sem usar a monotonicidade total das matrizes descritas anteriormente.

3.2.2 Algoritmo de União de GDAGs

O procedimento para calcular todos os melhores caminhos entre vértices de $origs_S$ e vértices de $dest_I$ é dado a seguir. Ele é baseado em um algoritmo originalmente publicado em [6], de natureza recursiva e proposto para o modelo PRAM. A apresentação que fazemos agora é bastante diferente, assim como a análise da complexidade temporal, visando facilitar a descrição do algoritmo paralelo da Seção 3.4.

A idéia central deste algoritmo é iniciar cada etapa com certos vértices *marcados* em $origs_S$ e $dest_I$, sendo que todos os caminhos ótimos entre vértices marcados já foram determinados. A cada etapa o número de pontos marcados dobra através de várias varreduras.

Algoritmo 3.2: **Melhores Caminhos de $origs_S$ a $dest_I$.**
Entrada: $AL'_S(t \times k)$ e $AL'_I(k \times t)$.
Saída: $AL'_U(t \times t)$ e $M_U(t \times t)$.

1 $M_U(0,0) \leftarrow 0$,
$M_U(0,t-1) \leftarrow 0$,
$M_U(t-1,0) \leftarrow -1$,
$M_U(t-1,t-1) \leftarrow k-1$

2 $AL'_U(0,0) \leftarrow AL'_S(0,0) + AL'_I(0,0)$,
$AL'_U(0,t-1) \leftarrow AL'_S(0,0) + AL'_I(0,t-1)$,
$AL'_U(t-1,t-1) \leftarrow AL'_S(t-1,k-1) + AL'_I(k-1,t-1)$,
$AL'_U(t-1,0) \leftarrow -\infty$

3 Marque $origs_S(0)$, $origs_S(t-1)$, $dest_I(0)$, $dest_I(t-1)$

4 $x \leftarrow \lceil \log_2(t-1) \rceil$

5 Repita

5.1 $x \leftarrow x - 1$

5.2 Para todo vértice $origs_S(q)$ não marcado com q múltiplo de 2^x calcule os caminhos para todos os pontos marcados em $dest_I$. Cada vértice marcado em $dest_I$ lidera uma varredura. Marque os vértices usados de $origs_S$.

5.3 Para todo vértice $dest_I(q)$ não marcado com q múltiplo de 2^x calcule os caminhos a partir de todos os pontos marcados em $origs_S$ (incluindo os marcados nesta iteração do laço). Cada vértice marcado em $origs_S$ lidera uma varredura. Marque os vértices usados de $dest_I$.

6 Até que $x = 0$.

fim do algoritmo.

O algoritmo anterior foi descrito para a união de dois GDAGs alinhados verticalmente (união vertical). Como já comentamos, para o caso de GDAGs alinhados horizontalmente (união horizontal) o procedimento é análogo.

Teorema 3.1 *O Algoritmo 3.2 calcula corretamente as matrizes AL'_U e M_U a partir de AL'_S e AL'_I.*

Prova. Vamos provar que no início do laço 5 e ao final de cada iteração temos como invariante a seguinte afirmação: se $origs_S(i)$ e $dest_I(j)$ estão marcados, então $AL'_U(i,j)$ e $M_U(i,j)$ já foram calculados. A demonstração é indutiva, partindo do fato de que no início apenas os extremos de $origs_S$ e $dest_I$ estão marcados.

O passo 5.2 executa uma varredura para cada vértice de $dest_I$, determinando os caminhos deste vértice para alguns ainda não marcados em $origs_S$. Os vértices $origs_S(q)$ por marcar são tais que q é múltiplo de 2^x, os já marcados são tais que q é múltiplo de 2^{x+1}. Desta forma, os vértices já marcados e os por marcar se intercalam em $origs_S$.

Isto garante que a varredura pode ser feita como indicado no Algoritmo 3.1. Com a marcação dos novos vértices de $origs_S$, o invariante se mantém.

Deve-se notar que em alguns momentos, dependendo do valor de $t-1$, o último trecho da varredura não cobrirá nenhum novo ponto de $origs_S$. Além disso, o algoritmo não faz menção aos dados não significativos de AL'_U (ver Figura 3.3) e como evitá-los nas varreduras, mas nada disto é significativo para a certificação ou análise temporal do algoritmo.

O passo 5.3 pode ser analisado de maneira semelhante, comprovando o invariante. Finalmente, como o laço é feito até que $x = 0$, ou seja, até que todos os pontos sejam marcados, temos a certificação do algoritmo. □

Para calcular o tempo de execução do algoritmo usaremos as funções definidas a seguir. Estas funções serão usadas no algoritmo paralelo da Seção 3.4.

Definição 3.2 (função w) *A função w indica qual é a maior potência de 2 que divide o argumento:*

$$w : \mathbb{N}^+ \to \mathbb{N},\ w(n) = \max\{x \mid n \equiv 0 \pmod{2^x}\}\ .$$

Um vértice $origs_S(n)$ ou $dest_I(n)$ é marcado pelo Algoritmo 3.2 na iteração em que $x = w(n)$. $origs_S(n)$ começa a liderar varreduras já nesta iteração, enquanto $dest_I(n)$ irá liderar varreduras na iteração seguinte. Como já foi comentado, cada varredura tem duração $O(k + r)$, onde r é o

número de intervalos que a varredura considera, igual a $\lfloor \frac{t}{2^x} \rfloor$. Vamos estimar o tempo de cada varredura com base na seguinte hipótese: o tempo de cada varredura é proporcional ao número de acessos realizados às matrizes AL_S e AL_I[1]. Isto indica que o tempo de uma varredura seja proporcional a $k + \lfloor \frac{t}{2^x} \rfloor$.

Com estas considerações, podemos prosseguir para as definições seguintes.

Definição 3.3 (funções v_d e v_o)

1. *$v_d(n,t,k)$ indica, a menos de uma constante multiplicativa, qual é o tempo total das varreduras lideradas pelo vértice $dest_I(n)$ quando são dados k (número de possíveis caminhos a serem testados) e t (número de vértices de origem). Este custo é dado pela somatória $\sum_{x=0}^{w(n)-1} \left(k + \lfloor \frac{t}{2^x} \rfloor\right)$. Considerando a somatória $\sum_{x=0}^{w(n)-1} \left(k + \frac{t}{2^x}\right)$ no lugar da somatória original, o erro cometido é inferior a $w(n) \leq \log t$. Assim sendo, definimos*

$$v_d : (\mathbb{N}^+ \times \mathbb{N}^+ \times \mathbb{N}^+) \to \mathbb{N},\ v_d(n,t,k) = \sum_{x=0}^{w(n)-1} \left(k + \frac{t}{2^x}\right) = w(n)k + 2t - \left\lfloor \frac{t}{2^{w(n)-1}} \right\rfloor .$$

2. *De maneira análoga, $v_o(n,t,k)$ indica o tempo total das varreduras lideradas pelo vértice $orig_S(n)$. Este tempo é dado por $\sum_{x=0}^{w(n)} \left(k + \lfloor \frac{t}{2^x} \rfloor\right)$ mas, por razões semelhantes às apresentadas anteriormente, usaremos a seguinte definição:*

$$v_o : (\mathbb{N}^+ \times \mathbb{N}^+ \times \mathbb{N}^+) \to \mathbb{N},\ v_o(n,t,k) = \sum_{x=0}^{w(n)} \left(k + \frac{t}{2^x}\right) = (w(n)+1)k + 2t - \left\lfloor \frac{t}{2^{w(n)}} \right\rfloor .$$

É importante notar que as funções v_d e v_o serão usadas não apenas na análise dos algoritmos deste capítulo, mas também na *implementação* do algoritmo paralelo da Seção 3.4. O algoritmo paralelo as utiliza para determinar o tempo necessário para resolução de certos subproblemas e distribuí-los entre os processadores.

Deve-se notar também que estas definições são flexíveis, no sentido de que permitem que se considere apenas um número limitado de vértices de destino ou origem (parâmetro t) e um número limitado de vértices de $meio_U$ (parametro k).

A seguinte função também será usada.

Definição 3.4 (função W)

$$W : \mathbb{N}^+ \to \mathbb{N},\ W(n) = \sum_{i=1}^{n} w(i)$$

Lema 3.2 $n - 2 - \lfloor \log n \rfloor \leq W(n) \leq n - 1$.

[1] Esta estimativa é comentada no final deste capítulo

Prova. Usando a notação de Iverson [21], indicamos por $[i \equiv 0 \pmod{2^j}]$ o valor 1 se i é múltiplo de 2^j, 0 caso contrário. Desta maneira, $w(i) = \sum_{j=1}^{\lfloor \log n \rfloor} [i \equiv 0 \pmod{2^j}]$ e assim

$$W(n) = \sum_{i=1}^{n} w(i) = \sum_{i=1}^{n} \sum_{j=1}^{\lfloor \log n \rfloor} [i \equiv 0 \pmod{2^j}] = \sum_{j=1}^{\lfloor \log n \rfloor} \sum_{i=1}^{n} [i \equiv 0 \pmod{2^j}] = \sum_{j=1}^{\lfloor \log n \rfloor} \left\lfloor \frac{n}{2^j} \right\rfloor .$$

Desta maneira, temos os seguintes limites superior e inferior:

$$\begin{aligned} W(n) &\leq \sum_{j=1}^{\lfloor \log n \rfloor} \frac{n}{2^j} = n - \frac{n}{2^{\lfloor \log n \rfloor}} \leq n - 1 \\ W(n) &\geq \sum_{j=1}^{\lfloor \log n \rfloor} \left(\frac{n}{2^j} - 1 \right) = n - \frac{n}{2^{\lfloor \log n \rfloor}} - \lfloor \log n \rfloor > n - 2 - \lfloor \log n \rfloor \end{aligned}$$

□

O Teorema 3.3 a seguir indica o tempo de execução do Algoritmo 3.2 em função de k e t. Pela definição de k e t temos que $k = O(t)$, mas o algoritmo pode ser facilmente adequado para calcular caminhos envolvendo *trechos* de $orig_S$ e $dest_I$, caso em que a relação entre k e t não é clara. A análise feita a seguir é extensível para casos como este.

Teorema 3.3 *O Algoritmo 3.2 tem tempo de execução* $\Theta(kt + t^2)$ *e utiliza espaço* $\Theta((k+t)^2)$, *onde* k *é o número de vértices considerados de* $meio_U$ *e* t *o número de vértices considerados de* $orig_S$ *e de* $dest_I$.

Prova. O espaço utilizado é determinado pelas matrizes AL_S, AL_I e AL_U. Note que AL_U ocupa aproximadamente o mesmo espaço que AL_S e AL_I ocupavam juntas. Na verdade, há uma discrepância que é compensada quando quatro GDAGs são unidos em um.

O tempo total é dado pela totalização dos custos de todas as varreduras lideradas por todos vértices de $orig_S$ e $dest_I$. Estes custos já foram mencionados na Definição 3.3, menos para os extremos, que são envolvidos em todas as iterações do laço 5. Calculando primeiramente os custos para os vértices *interiores* de $orig_S$ e $dest_I$ temos

$$\begin{aligned} T(t,k) = \quad & \Theta \left(\sum_{i=1}^{t-2} v_d(i,t,k) + \sum_{i=1}^{t-2} v_o(i,t,k) \right) = \\ & \Theta \left(\sum_{i=1}^{t-2} \left(w(i)k + 2t - \frac{t}{2^{w(i)-1}} \right) + \sum_{i=1}^{t-2} \left((w(i)+1)k + 2t - \frac{t}{2^{w(i)}} \right) \right) = \\ & \Theta \left(k \sum_{i=1}^{t-2} (2w(i)+1) + t \sum_{i=1}^{t-2} \left(4 - \frac{3}{2^{w(i)}} \right) \right) = \Theta(kt + t^2) \ . \end{aligned}$$

Note que na última passagem usamos o Lema 3.2 para resolver a primeira somatória ($\Theta(t)$). Na segunda somatória pode-se desprezar o termo $\frac{3}{2^{w(i)}}$.

Estes cálculos, assim como o próprio Algoritmo 3.2, não consideram que algumas varreduras podem ser feitas em um trecho de $meio_U$ de tamanho menor do que k, não considerando caminhos inexistentes no GDAG. Os presentes resultados não seriam alterados por estas considerações.

As varreduras lideradas pelos extremos de $orig_S$ e $dest_I$ tomam tempo $O(kt + t^2)$ (pode-se demonstrar que este tempo é $O(k \log t + t))$, logo o Algoritmo 3.2 é executado em tempo $\Theta(kt + t^2)$. □

Um cálculo mais preciso é mostrado no Apêndice A, considerando as partes efetivamente usadas das matrizes. Os resultados lá apresentados estão de acordo com os que foram mostrados aqui.

3.3 Algoritmo Seqüencial para o ATS

A resolução do problema ATS pode ser feita através do uso do Algoritmo 3.2. Partindo de subgrafos 2×2, cujas matrizes AL podem ser calculadas em tempo $O(1)$, aplicamos o processo de união de GDAGs até obter a matriz do GDAG completo.

Vamos supor, sem perda de generalidade, que $n_b \geq n_a$. Isto significa que haverá mais etapas de uniões horizontais do que de verticais. Vamos supor também, por simplicidade, que $n_a = 2^a$ e $n_b = 2^b$ com $b \geq a$ inteiros. Se este não for o caso, pode-se aumentar as cadeias envolvidas sem que a complexidade assintótica do problema se altere.

Lema 3.4 *Após v etapas de união vertical e h etapas de união horizontal teremos $2^{a+b-v-h}$ GDAGs, todos com dimensões $(2^v + 1) \times (2^h + 1)$*

Prova. Simples, usando indução finita. Com $v = h = 0$ temos os 2^{a+b} GDAGs originais (2×2). Uma etapa de união vertical reduz o número de GDAGs à metade, sendo que o número de linhas dos GDAGs resultantes é igual à soma dos números de linhas dos GDAGs fundidos, menos um (devido à fronteira comum). As uniões horizontais operam de modo análogo. Note que com $v = a$ e $h = b$ obtemos o GDAG completo com dimensões $(n_a + 1) \times (n_b + 1)$. □

Teorema 3.5 *O problema ATS para cadeias de tamanho n_a e n_b, $n_b \geq n_a$ pode ser resolvido em tempo $O(n_a n_b \log n_a + n_b^2)$ e espaço $O(n_b^2)$.*

Prova. Antes das primeiras etapas de união, todas as matrizes AL associadas aos GDAGs elementares podem ser determinadas em tempo $\Theta(2^{a+b}) = \Theta(n_a n_b)$

Após v etapas de união vertical e h etapas de união horizontal, pelo Lema 3.4 a dimensão de cada GDAG será $(2^v + 1) \times (2^h + 1)$, o que resulta em matrizes AL de $(2^v + 2^h + 1)^2$ elementos. O espaço usado em memória é

$$\Theta(2^{a+b-v-h}(2^v + 2^h + 1)^2) = \Theta(2^{a+b}(2^{v-v} + 2^{h-v} + 2^{-v})(2^{v-h} + 2^{h-h} + 2^{-h})) = \\ \Theta(2^{a+b}(1 + 2^{h-v})(1 + 2^{v-h})) = \Theta(2^{a+b} 2^{|h-v|}) \ .$$

O espaço utilizado pode ser limitado fazendo com que $|h - v|$ seja mínimo ao longo do processo, intercalando uniões verticais e horizontais. No final, uma seqüência de uniões horizontais faz esta diferença subir para $b - a$. Desta maneira, o espaço aumenta até $\Theta(2^{2b}) = \Theta(n_b^2)$.

Supondo que as etapas de uniões verticais e horizontais se intercalam nas primeiras etapas, numa etapa de uniões verticais teremos $v = h$ e o tempo necessário, segundo o Lema 3.4 será $\Theta((2^v + 2^h + 1)^2) = \Theta(2^{2v})$. Há $2^{a+b-v-h} = 2^{a+b-2v}$ GDAGs para unir dois a dois, logo o trabalho total numa etapa de uniões verticais é $\Theta(2^{a+b})$.

Numa etapa de uniões horizontais teremos $v = h + 1$. Analogamente ao que ocorre numa etapa de uniões verticais, o tempo de execução é $\Theta(2^{a+b})$.

Assim, após $v = h = a$ etapas de uniões horizontais e verticais intercaladas, o tempo de execução total é $\Theta(a2^{a+b})$.

Nas últimas $b - a$ etapas de uniões horizontais, quando nenhuma união vertical é necessária, cada união levará tempo $\Theta((2^a + 2^h + 1)^2) = \Theta(2^{2h})$, uma vez que $h > a$. Como numa etapa há $2^{b-h} =$ GDAGs para unir aos pares, cada etapa levará tempo $\Theta(2^{b+h})$. O tempo total destas últimas etapas é

$$\Theta\left(\sum_{i=a+1}^{b} 2^{b+h}\right) = \Theta(2^{2b} - 2^{b+1-a+1}) = \Theta(2^{2b} - 2^{a+b}) \ .$$

O tempo total para resolução do ATS é

$$\Theta(a2^{a+b} + 2^{2b} - 2^{a+b}) = \Theta(a2^{a+b} + 2^{2b}) =$$
$$\Theta(n_a n_b \log n_a + n_b^2) \ .$$

□

Alternativamente, este problema pode ser resolvido em tempo $\Theta(n_a n_b \log n_b)$ pelas técnicas apresentadas em [40].

3.4 Algoritmo Paralelo para o ATS

O algoritmo paralelo, como já mencionado, começa com a divisão do GDAG em p subgrafos, como ilustrado na Figura 3.2 (página 35). Cada subgrafo $G_{1,l,c}$, $1 \leq l \leq \sqrt{p}$, $1 \leq c \leq \sqrt{p}$ tem sua matriz $AL_{G_{1,l,c}}$ calculada por um processador, ocupando espaço $O\left(\frac{(n_a+n_b)^2}{p}\right)$ na memória local deste processador.

À medida em que as etapas de união se sucedem e os subgrafos aumentam, as matrizes de alinhamento também aumentam, evidenciando a necessidade de distribuir estas matrizes entre os processadores. Vamos definir primeiramente como esta distribuição é feita.

3.4.1 Distribuição das Matrizes AL

O tamanho da matriz AL resultante da união de dois GDAGs não difere muito da soma dos tamanhos das matrizes dos dois GDAGs originais. Na pior das hipóteses, o tamanho da nova matriz é o dobro da soma do tamanho das matrizes originais, quando consideramos apenas a parte "útil" destas matrizes, mas esta discrepância desaparece na etapa de união seguinte. O GDAG formado pela união de quatro GDAGs tem uma matriz AL de tamanho igual à soma das matrizes dos quatro GDAGs originais.

Quando a matriz AL de um GDAG é gerada a partir das matrizes dos GDAGs constituintes, ela deve ser distribuída pelas memórias locais dos processadores que guardavam as matrizes dos GDAGs originais. Isto significa que na u-ésima etapa de união, cada matriz estará distribuída inicialmente nas memórias locais de 2^{u-1} processadores. Duas matrizes serão usadas no cálculo de uma nova matriz a ser distribuída nas memórias de 2^u processadores. Estes processadores farão o cálculo da nova matriz em paralelo.

Por razões que se tornarão aparentes em breve, as matrizes serão divididas em faixas de linhas *ou* colunas para distribuição entre os processadores. A escolha da forma da divisão depende do papel que o GDAG correspondente terá na próxima fase de união.No início do processo de união de dois GDAGs alinhados verticalmente, a matriz do GDAG *superior* deve estar distribuída por *colunas*, enquanto que a matriz do GDAG *inferior* deve estar distribuída por *linhas.* Numa etapa de união entre GDAGs alinhados horizontalmente, as matrizes dos GDAGs esquerdo e direito devem estar distribuídas por colunas e por linhas, respectivamente.

Vamos ilustrar a união usando o caso da união vertical de um GDAG superior S com um inferior I para gerar o GDAG U (ver Figura 3.3). A partir deste ponto, vamos considerar apenas a parte útil (para a união em questão) das matrizes de alinhamento destes GDAGs.

Sendo q o número de processadores associados a um certo GDAG, a sua matriz AL' será dividida em $2q$ faixas de linhas ou colunas contíguas, cada processador recebendo duas faixas. Isto é suficiente para que haja um certo equilíbrio na distribuição de dados, um número maior de faixas prejudicaria a eficiência do algoritmo. A parte da matriz AL que não será usada nesta etapa de união deve ser distribuída de maneira semelhante. Não iremos nos preocupar com esta parte até o final da união, quando a nova matriz AL_U gerada é redistribuída entre os processadores.

Seja PS o conjunto dos q processadores $PS_1, PS_2, \ldots, PS_q$ que guardam AL_S e PI o conjunto dos q processadores $PI_1, PI_2, \ldots, PI_q$ que guardam AL_I. Cada um destes $2q$ processadores está associado a alguns vértices de $meio_U$, ou seja,

- $PS_i, 1 \leq i \leq q$ guarda os pesos dos melhores caminhos de todos os vértices de $orig_S$ para alguns vértices de $meio_U$.
- $PI_i, 1 \leq i \leq q$ guarda os pesos dos melhores caminhos de alguns vértices de $meio_U$ para todos os vértices de $dest_I$.

Os vértices de $meio_U$ a que um certo processador está associado estão divididos em dois

intervalos distintos.

Veremos como estes $2q$ processadores fazem o cálculo da nova matriz em paralelo.

3.4.2 Divisão do Problema da União de GDAGs

Cada processador será responsável pela geração de uma parte da matriz AL'_U. A solução natural para este problema seria dividir AL'_U em linhas ou colunas, de preferência já de acordo com a distribuição esperada no final da união, mas infelizmente a distribuição de carga de trabalho não seria equilibrada e muitos dados teriam que ser replicados para uso por diversos processadores.

Uma divisão de AL'_U em blocos é mais adequada. Suponhamos que AL'_U seja dividida em $4q^2$ blocos, organizados em $2q$ linhas e $2q$ colunas. Cada bloco representa um subproblema a ser resolvido. Naturalmente, alguns destes blocos não precisam ser calculados pois envolvem caminhos inexistentes, mas isto não afeta os resultados que iremos mostrar a seguir.

As seqüências $orig_S$ e $dest_I$ devem ser divididas em $2q$ segmentos que se sobrepõem apenas nas extremidades. Sendo t' o comprimento de cada segmento, devemos ter $t' = \frac{t-1}{2q} + 1$, que iremos supor que é inteiro por simplicidade.

Se cada bloco tivesse que ser calculado separadamente, seria necessário utilizar uma faixa de tamanho $t' \times k$ de AL'_S e uma de tamanho $k \times t'$ de AL'_I. No entanto, se resolvermos o problema para as *bordas* do bloco, podemos reduzir a quantidade de dados utilizada no subproblema e, portanto, o tempo de processamento.

A Figura 3.4 mostra como isto pode ser feito. A figura mostra as partes de AL'_S e AL'_I necessárias para calcular um bloco de dados de AL'_U, mais exatamente os valores de $AL'_U(i,j)$ para $o_1 \leq i \leq o_2$ e $d_1 \leq j \leq d_2$. Nos vértices dos GDAGs mostrados, os rótulos se referem aos índices nas seqüências $orig_S$, $dest_I$ e $meio_U$.

Em AL'_S temos uma faixa entre (e incluindo) as linhas o_1 e o_2. Para cada linha nesta faixa, digamos para um certo i, $o_1 \leq i \leq o_2$, os únicos pontos de interesse em $meio_U$ são os que estão entre $M_U(i,d_1)$ e $M_U(i,d_2)$, de acordo com a Propriedade 3.2. Desta maneira, só os dados entre $AL'_S(i,d_1)$ e $AL'_S(i,d_2)$ são úteis na linha i. Estes limites variam de maneira não decrescente com i, formando fronteiras irregulares à esquerda e à direita dos dados que serão necessários.

Em AL'_I, por razões semelhantes (baseadas na Propriedade 3.1), teremos uma faixa útil entre as colunas d_1 e d_2, com bordas irregulares acima e abaixo dos dados necessários.

Dois blocos distintos e disjuntos (a menos das bordas) de AL'_U terão regiões úteis em AL'_S e AL'_I também disjuntas (a menos das bordas). Este fato, facilmente demonstrável pelas Propriedades 3.1 e 3.2, é muito importante porque faremos uma distribuição do cálculo dos blocos de AL'_U entre os processadores. Uma sobreposição dos blocos de dados a serem utilizados implicaria em replicação de dados entre processadores e maior tempo de execução total.

A irregularidade das bordas das regiões úteis para cada bloco pode tornar mais difícil a implementação do algoritmo paralelo, mas não aumenta a complexidade assintótica, como veremos mais à frente. A alternativa seria criar blocos retangulares, usando como limites os pontos

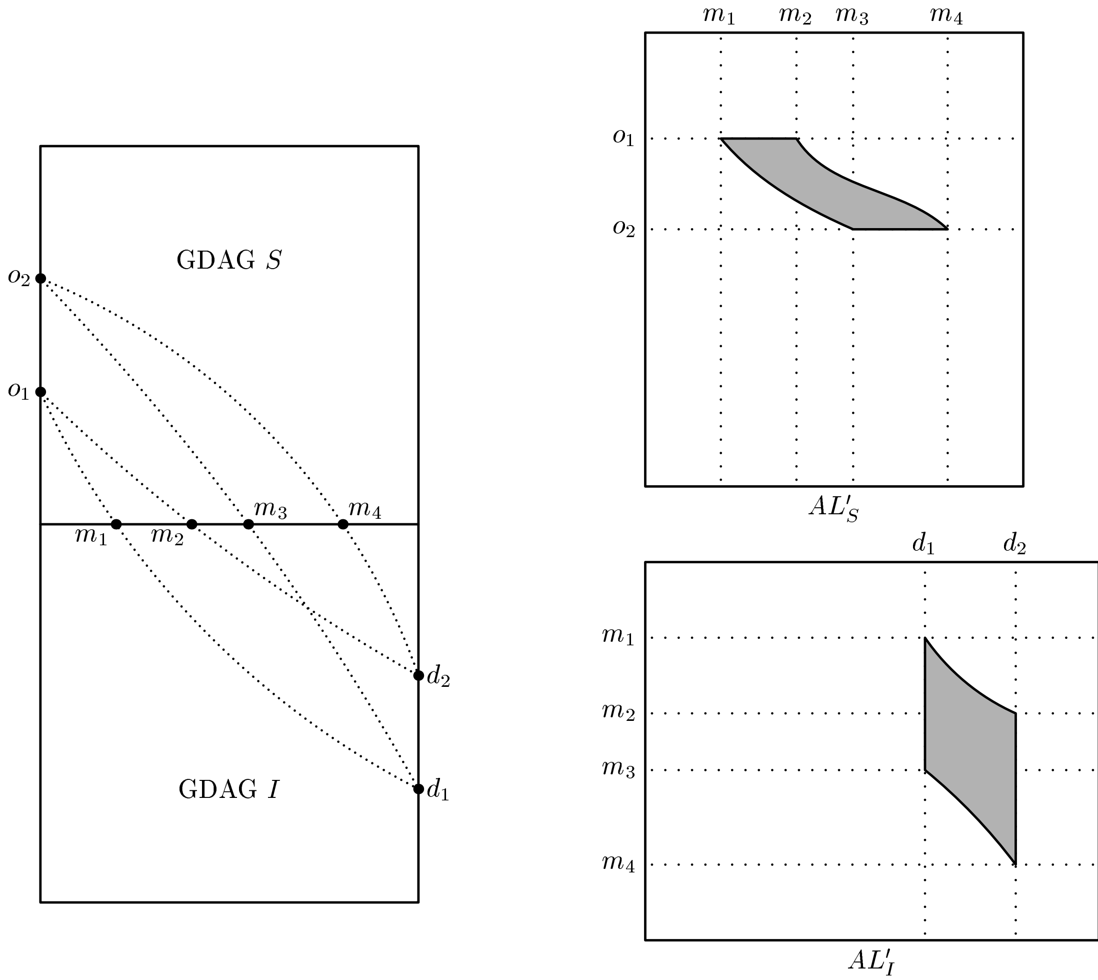

Figura 3.4: Dados necessários para o cálculo de um bloco de AL'_U.

extremos da região efetiva. Por exemplo, na Figura 3.4 poderíamos usar a submatriz de AL'_S entre as linhas o_1 e o_2 e as colunas m_1 e m_4. O custo disso seria permitir a sobreposição das regiões úteis.

3.4.3 Determinação dos Subproblemas

A determinação das bordas das regiões úteis para todos os subproblemas envolve a determinação de $M_U(i,j)$ para todos os pares (i,j) em que i ou j é múltiplo de $t'-1$. A determinação destas linhas e colunas regularmente espaçadas em M_U pode ser feita em paralelo conforme o lema abaixo.

Lema 3.6 *Todos os valores de $AL'_U(i,j)$ e $M_U(i,j)$ com i ou j múltiplo de $t'-1$ podem ser calculados em paralelo pelos $2q$ processadores que guardam AL_S e AL_I em tempo $O\left(k \log \frac{t}{q} + qt\right)$*

e espaço $O\left(\frac{t^2}{q}\right)$ *com duas rodadas de comunicação em que* $O(qt)$ *dados são transmitidos ou recebidos por cada processador.*

Prova. Os processadores de PS (que guardam AL'_S) irão determinar as colunas selecionadas de AL'_U, enquanto que os processadores de PI (que guardam AL'_I) determinarão as linhas. Vamos nos concentrar na determinação das linhas (a determinação das colunas é análoga).

Um determinado processador PI_i, $1 \leq i \leq q$ guarda os melhores caminhos de $\frac{k}{q}$ vértices de $meio_U$ para todos os t vértices de $dest_I$. Ele deve receber os pesos dos melhores caminhos dos $2q+1$ vértices selecionados de $orig_S$ para os mesmos vértices de $meio_U$, o que envolve a transferência de $O(k)$ dados dos processadores $PS_1, PS_2, \ldots, PS_q$. Dependendo dos detalhes de distribuição das matrizes pelos processadores, é possível que apenas o processador PS_i precise enviar dados para PI_i, mas isto não é importante para esta análise. Ao mesmo tempo, cada processador de PS precisa receber $O(k)$ dados dos processadores de PI. Todas estas transferências podem ser feitas em apenas uma rodada de comunicação que envolve o envio/recebimento de $O(k)$ dados por processador.

O processador PI_i determina então os pesos dos melhores caminhos da amostra de $orig_S$ ($2q+1$ vértices) para $dest_I$ (t vértices) *que passam pelos* $\frac{k}{q}$ *vértices de* $meio_U$. Inicialmente, o Algoritmo 3.2 é usado para encontrar os caminhos entre os vértices selecionados de $orig_S$ e os vértices selecionados de $dest_I$. Pelo Teorema 3.3 o tempo de execução desta etapa é $\Theta\left(\frac{k}{q}(2q+1) + (2q+1)^2\right) = \Theta(k+q^2)$.

Após isto, os caminhos para os demais vértices de $dest_I$ são determinados através de $\lceil \log t' \rceil$ etapas em que se faz uma varredura para cada vértice selecionado de $orig_S$. Cada varredura tem duração $\Theta\left(\frac{k}{q} + r\right)$ sendo que r dobra a cada etapa, de $2q+1$ a t. O tempo total desta fase é $\Theta\left((2q+1)\left(\frac{k}{q}\log t' + t\right)\right) = \Theta\left(k \log \frac{t}{q} + qt\right)$.

Após esta etapa, para cada par de vértices contendo um vértice de $dest_I$ e outro da amostra de $orig_S$ teremos q candidatos para o melhor caminho. Cada candidato foi gerado por um dos processadores de PI, com informações de peso total e vértice de $meio_U$ utilizado. Para determinar o melhor candidato para cada caminho, dividimos $dest_I$ em q intervalos e atribuímos o i-ésimo intervalo a PI_i, que deve receber todos os candidados a melhor caminho entre a amostra de $orig_S$ e seu intervalo de $dest_I$. Isto envolve uma rodada de comunicação em que cada processador envia/recebe $O\left(q(2q+1)\frac{t}{q}\right) = O(qt)$ dados.

A pesquisa pelos melhores caminhos pode ser feita em tempo $O(qt)$ com um procedimento de busca simples. Isto pode ser melhorado, mas não altera o resultado do lema, visto que a própria transferência dos dados a serem manipulados leva tempo $\Theta(qt)$.

Para concluir, destacamos que o espaço em memória utilizado é dominado pelo espaço necessário para o armazenamento da matriz AL'_I (ou AL'_S).

□

Várias melhorias podem ser aplicadas ao procedimento descrito acima, sem impacto na complexidade assintótica mas úteis na implementação do algoritmo. Uma delas é distribuir os

candidatos a melhor caminho de uma maneira que leve em consideração o número efetivo de candidatos: nem todos os pares de vértices de $orig_S$ e $dest_I$ admitem caminhos em todos os intervalos de $meio_U$ (alguns pares não admitem qualquer caminho).

Outra possível melhoria envolve o procedimento de seleção dos melhores candidatos. Se cada processador receber a informação de qual trecho de $meio_U$ foi usado por cada candidato, é possível usar os trechos de $meio_U$ da mesma maneira que usamos os vértices de $meio_U$ nas *varreduras* definidas no Algoritmo 3.1. Isto significa que novamente podemos usar uma variação do Algoritmo 3.2, desta vez para achar os melhores caminhos de $2q + 1$ vértices de $orig_S$ para t/q vértices de $dest_I$ através de $2q$ trechos de $meio_U$. Uma análise semelhante à realizada na demonstração anterior nos leva a um tempo $O\left(q^2 \log \frac{t}{q^2}\right)$, no lugar do tempo $O(qt)$ mencionado.

Ao fim do procedimento descrito anteriormente, cada processador terá informações sobre $4q$ dos $4q^2$ subproblemas (novamente, estamos ignorando o fato de que alguns destes subproblemas são vazios, sem prejuízo para a análise assintótica). Os processadores $PS_1, PS_2, \ldots, PS_q$ terão as informações sobre as bordas das regiões úteis de AL'_S, ou seja, os valores de $M_U(i, j)$ para todo i e todo j entre 1 e t com j múltiplo de $t' - 1$. $PI_1, PI_2, \ldots, PI_q$ terão papel semelhante com as informações sobre o uso de AL'_I.

Estas informações sobre as *bordas* dos subproblemas são então distribuídas entre todos os processadores, para que eles possam definir quais dos dados que eles armazenam estão associados a quais subproblemas. Cada processador envia $\Theta(qt') = \Theta(t)$ dados e recebe $O(qt)$ dados no pior caso.

3.4.4 Previsão de Custos de um Subproblema

Dado um subproblema em particular, as informações relativas às suas bordas estarão disponíveis em dois processadores: um de PS, para as bordas em AL'_S, outro de PI, para as bordas em AL'_I. Estas informações permitem calcular tanto o espaço quanto o tempo necessários para a resolução do subproblema. Isto é exposto e demonstrado nos lemas 3.7 e 3.8. O algoritmo para resolução de um subproblema, uma modificação do Algoritmo 3.2, é exposto a seguir.

Algoritmo 3.3:	**Cálculo de um bloco de AL'_U (Melhores Caminhos dos vértices de $orig_S(i') \ldots orig_S(i' + t')$ aos vértices de $dest_I(j') \ldots dest_I(j' + t')$).**
Entrada:	$M_U(i, j)$ para todo par (i, j) na borda do bloco, partes necessárias de AL'_S e AL'_I.
Saída:	blocos $t' \times t'$ de AL'_U e de M_U.

1 Marque $orig_S(i')$, $orig_S(i' + t')$, $dest_I(j')$ e $dest_I(j' + t')$ (o problema já foi resolvido para as bordas do bloco)

2 $x \leftarrow \lceil \log_2 t' \rceil$

3 Repita

3.1 $\quad x \leftarrow x - 1$

3.2 Para todo vértice $orig_S(q)$ não marcado com $q - i'$ múltiplo de 2^x calcule os caminhos para todos os pontos marcados em $dest_I$. Cada vértice j marcado em $dest_I$ (com exceção dos extremos) lidera uma varredura usando um trecho de $meio_U$ de comprimento $M_U(i' + t', j) - M_U(i', j) + 1$. Marque os vértices usados de $orig_S$.

3.3 Para todo vértice $dest_I(q)$ não marcado com $q - i'$ múltiplo de 2^x calcule os caminhos a partir de todos os pontos marcados em $orig_S$ (incluindo os marcados nesta iteração do laço). Cada vértice i marcado em $orig_S$ (com exceção dos extremos) lidera uma varredura usando um trecho de $meio_U$ de comprimento $M_U(i, j' + t') - M_U(i, j') + 1$. Marque os vértices usados de $dest_I$.

4 Até que $x = 0$.

fim do algoritmo.

Lema 3.7 *O espaço necessário para determinação do bloco de AL'_U iniciado em (i', j'), que chamaremos $ESub_{i',j'}$, pode ser determinado em tempo $\Theta(t') = \Theta\left(\frac{t}{q}\right)$. O espaço total para cálculo de todos os blocos de AL'_U é $\Theta(t^2)$.*

Prova. O espaço necessário para cada subproblema é dado pelo tamanho do bloco a ser gerado ($\Theta\left(\frac{t^2}{q^2}\right)$, fixo) e pelas áreas previamente delimitadas das matrizes AL'_S (t' trechos de linhas) e AL'_I (t' trechos de colunas). A totalização de cada uma destas áreas é bastante simples, podendo ser feita separadamente pelos dois processadores que mantém as informações. Uma comunicação de tamanho $O(1)$ basta para reunir estes resultados. □

Lema 3.8 *O tempo necessário para cálculo de um bloco de AL'_U, que chamaremos $TSub_{i',j'}$ para um bloco iniciado em (i', j'), pode ser determinado em tempo $\Theta(t') = \Theta\left(\frac{t}{q}\right)$.*

Prova. Como mencionado na demonstração do Teorema 3.3, o tempo de execução do Algoritmo 3.3 é determinado pelo custo total de suas varreduras. Note que as varreduras agora operam sobre trechos de tamanhos variáveis de $meio_U$, tendo, portanto, tempos de execução variados.

Usando as funções v_d e v_o (Definição 3.3) podemos calcular uma estimativa para o tempo de execução do subproblema, $TSub_{i',j'}$. A estimativa é válida a menos de uma constante multiplicativa.

$$\begin{aligned} TSub_{i',j'} = & \sum_{j=1}^{t'-1} v_d(j, t', M_U(i' + t', j) - M_U(i', j) + 1) \\ + & \sum_{i=1}^{t'-1} v_o(i, t', M_U(i, j' + t') - M_U(i, j') + 1) \end{aligned}$$

Uma vez que v_d e v_o podem ser avaliadas em tempo constante, a estimativa anterior pode ser obtida em tempo $\Theta(t') = \Theta\left(\frac{t}{q}\right)$. Na verdade, dois processadores devem realizar o cálculo para determinado subproblema, cada um resolvendo uma somatória. A totalização final será explicada a seguir. □

Pelos lemas anteriores, cada processador pode realizar cálculos sobre $4q$ subproblemas em tempo $\Theta(qt') = \Theta(t)$. Todos os $\Theta(q^2)$ dados gerados podem ser enviados em uma rodada de comunicação para um processador, digamos PS_1, que realiza as totalizações de estimativas de tempo e espaço para todos os processadores. Este processador deve então distribuir os subproblemas para os processadores.

3.4.5 Escalonamento dos Subproblemas aos Processadores

Para estudar o escalonamento, vamos primeiro verificar alguns fatos sobre o tempo e o espaço necessários para os subproblemas.

A soma dos tempos de todos os subproblemas é essencialmente igual ao tempo de processamento do Algoritmo 3.2, que pelo Teorema 3.3 é $\Theta(kt + t^2)$. Pelo Lema A.4, desenvolvido no Apêndice A, temos que a razão entre o *maior* tempo de execução possível para um subproblema e o tempo total de todos os subproblemas é inferior a $\frac{1}{2q}$. Quanto ao espaço, chegamos a conclusões semelhantes: o espaço total usado pelos subproblemas é essencialmente igual ao espaço necessário para o algoritmo seqüencial e, pelo Lema A.5 a razão entre o maior espaço necessário possível para um subproblema e o espaço total usado por todos os subproblemas é inferior a $\frac{5}{6q}$.

As demonstrações dos dois lemas mencionados anteriormente foram colocadas em um apêndice por serem muito extensas, podendo prejudicar a continuidade deste texto. É importante ressaltar que na demonstração destes lemas foram consideradas apenas as partes úteis das matrizes AL'_U, AL'_S e AL'_I.

O problema do escalonamento pode ser descrito nos seguintes termos: *dado um conjunto de subproblemas, para os quais se possui uma estimativa de tempo e espaço necessários, distribuí-los entre* $2q$ *processadores de tal forma que a memória utilizada por um processador esteja dentro de um limite* M *e o tempo de execução paralelo seja mínimo.*

Como o problema "básico" de escalonamento (que não considera limitações de memóra) é NP-difícil, encontrar a solução ótima para o problema descrito acima deve ser custoso. Temos a nosso favor o fato de que há no máximo $4q^2$ subproblemas para escalonar e consideramos q pequeno, mas ainda assim o uso de uma solução exponencial para o escalonamento será problemático por impor uma limitação adicional ao número de processadores que podem ser usados.

Na implementação real do algoritmo descrito neste capítulo podemos usar uma série de métodos distintos para realizar o escalonamento, talvez permitindo que o método a ser usado seja escolhido de acordo com as características do problema. Neste texto, estamos interessados em demonstrar que o escalonamento pode ser realizado de maneira aproximada, desde que o espaço e o tempo totais para um processador sejam $O(\frac{t^2}{q})$. Também avaliaremos o quanto a solução aproximada difere da solução ótima.

Nas demonstrações dos lemas A.4 e A.5 determinamos as seguintes estimativas para o total dos custos temporais e espaciais:

$$E(t,k) = 2tk - k^2 \quad (3.1)$$

$$T(t,k) = 2t^2 + 2tk \quad (3.2)$$

Estudando estas estimativas com $1 \leq k \leq t$ temos:

$$\frac{E(t,k)}{T(t,k)} \leq 0,268 \quad (3.3)$$

Para cada subproblema calcularemos um *custo* igual à soma dos custos temporal e espacial: $CSub_{i',j'} = TSub_{i',j'} + ESub_{i',j'}$. Dado um certo escalonamento, o *custo do escalonamento* é o maior custo total associado a um processador por este escalonamento. A soma dos custos de todos os subproblemas é obtido de 3.1 e 3.2:

$$C(t,k) = 2t^2 + 4tk - k^2 \quad (3.4)$$

O problema de escalonamento que deve ser resolvido passa a ser: *dado um conjunto de subproblemas para os quais se possui uma estimativa de custo, achar um escalonamento de custo mínimo destes subproblemas entre* $2q$ *processadores.* Temos então um problema de escalonamento *clássico*, para o qual há métodos de aproximação bem conhecidos.

O resultado a seguir dá um limite para o custo da solução ótima.

Lema 3.9 *$C_{opt}(t,k)$, o maior custo possível para um escalonamento ótimo, é tal que*

$$C_{opt}(t,k) \leq \frac{2,141 \cdot C(t,k)}{2q} .$$

Prova. Os lemas A.5 e A.4 indicam que o maior custo possível para um subproblema é $CSub_{max}(t,k) = \frac{5E(t,k)}{6q} + \frac{T(t,k)}{2q}$. Por 3.3 obtemos

$$CSub_{max}(t,k) \leq \left(\frac{5 \cdot 0,268}{6q \cdot 1,268} + \frac{1}{2q \cdot 1,268}\right) C(t,k) \leq \frac{1,141 \cdot C(t,k)}{2q} .$$

Em um escalonamento ótimo, a diferença entre o total dos custos associados a dois processadores distintos não pode superar $CSub_{max}$, caso contrário um escalonamento melhor poderia ser obtido pela transferência de um subproblema de um processador para outro. Assim temos

$$C_{opt}(t,k) \leq \frac{C(t,k)}{2q} + CSub_{max}(t,k) = \frac{2,141 \cdot C(t,k)}{2q} .$$

□

Com isto chegamos ao lema central desta seção:

Lema 3.10 *Os $4q^2$ subproblemas podem ser alocados aos $2q$ processadores de tal modo que o maior custo possível associado a um processador é $\frac{2{,}855 \cdot C(t,k)}{2q} = O(\frac{t^2+kt}{q})$. O tempo para determinação deste escalonamento é $\Theta(q^2 \log q)$.*

Prova. A seguinte heurística proposta por Graham [20, 24] encontra uma aproximação para o problema do escalonamento com custo inferior a $\frac{4C_{opt}(t,k)}{3}$: os subproblemas são alocados um a um, em ordem não crescente de custo, ao processador que está com a menor carga até o momento. Este é um resultado clássico cuja demonstração será omitida. Esta heurística pode ser aplicada em tempo $\Theta(q^2 \log q)$, determinado pela ordenação dos subproblemas.

O maior custo possível associado a um processador é dado diretamente pelo fator $\frac{4}{3}$ aplicado ao resultado do Lema 3.9. □

Considerando apenas o tempo de execução, o escalonamento acima garante que o tempo de execução paralelo para os subproblemas, que é o tempo de execução do processador que termina por último, é limitado da seguinte maneira:

$$TParalelo \leq CustoMax \leq \frac{C(k,t)}{2q} \cdot 2,855$$

Pela Equação 3.3 temos

$$TParalelo \leq \frac{E(k,t)+T(k,t)}{2q} \cdot 2,855 \leq \frac{1,268 \cdot T(k,t)}{2q} \cdot 2,855 \leq \frac{T(k,t)}{2q} \cdot 3,621 \ . \qquad (3.5)$$

Este fator de aproximação não é bom, embora seja suficiente para a demonstração dos resultados teóricos deste capítulo.

Este fator pode ser melhorado de várias formas, sendo que a mais óbvia é aumentar o número de subproblemas a serem resolvidos. Isto causa um impacto negativo em outras etapas do algoritmo, principalmente na determinação dos limites dos subproblemas e na quantidade de dados a serem enviados em cada rodada de comunicação. Os benefícios desta mudança podem ser pequenos, pois o fator de aproximação representa um pior caso possível, mas improvável. Para fins de aplicação prática, o número de subproblemas pode ser ajustado de maneira empírica para determinado ambiente de execução.

3.4.6 Distribuição e Resolução dos Subproblemas

Após determinado o escalonamento, o processador PS_1 usa uma rodada de comunicação para distribuir os q^2 dados sobre o escalonamento para os q processadores. Isto pode ser feito em uma rodada de comunicação de $O(q^3)$ dados. Usando mais rodadas de comunicação, com *broadcasts* parciais, podemos reduzir a quantidade de dados enviados por rodada. Por exemplo, com duas rodadas baixamos a quantidade de dados transmitida por processador para $O(q^{5/2})$. Por simplicidade, vamos considerar um único *broadcast*.

Tendo recebido os dados sobre o escalonamento, cada processador pode enviar seus dados para os processadores que irão usá-los. Esta etapa de comunicação é complexa devido à irregularidade das regiões de AL'_S e AL'_I necessárias, como mostrado na Figura 3.4. Os dados enviados

devem ser convenientemente rotulados para que os processadores que os recebem possam remontar as partes a serem usadas de maneira conveniente. Apesar disto, a distribuição pode ser feita em uma única rodada de comunicação em que cada processador recebe/envia $O\left(\frac{t^2+kt}{q}\right)$ dados.

De posse dos dados necessários, cada processador resolve localmente seus subproblemas em tempo $O\left(\frac{t^2+kt}{q}\right)$. Uma última etapa de comunicação é necessária para que os dados de AL_U sejam distribuídos de maneira adequada para a próxima etapa de união de GDAGs. Cada processador recebe/envia $O\left(\frac{t^2}{q}\right)$ dados. Deve-se lembrar que AL_U inclui partes originárias de AL_S e AL_I que não foram envolvidas no processo de união.

3.4.7 Resumo e Análise do Processo Paralelo de União de GDAGs

Temos então as seguintes etapas no processo de união de dois GDAGs:

1. Determinação dos subproblemas. Como já visto na Seção 3.4.3, isto pode ser feito em tempo $O\left(k\log\frac{t}{q}+qt\right)$, espaço $O\left(\frac{t^2}{q}\right)$ e três rodadas de comunicação em que $O(qt)$ dados são transmitidos ou recebidos por cada processador. A primeira rodada serve para distribuição dos dados necessários para o cálculo de certos caminhos no GDAG, enquanto que a segunda rodada distribui os valores destes caminhos entre os processadores para que os melhores sejam encontrados. Uma terceira rodada faz um *broadcast* das informações obtidas nesta etapa para todos os processadores.

2. Avaliação dos custos dos subproblemas. Como explicado na Seção 3.4.4, esta avaliação pode ser realizada em paralelo em tempo $O(t)$. Uma rodada de comunicação é necessária para centralizar as informações de custos ($O(q^2)$ dados) em um processador. Na verdade, esta rodada pode ser incorporada na terceira rodada do item anterior.

3. Escalonamento dos subproblemas. Como visto na Seção 3.4.5, o escalonamento pode ser determinado em tempo $O(q^2\log q)$. Um *broadcast* de tamanho total $O(q^3)$ sobre o escalonamento é feito para todos os processadores.

4. Distribuição e resolução dos subproblemas. Como visto na Seção 3.4.6, isto envolve duas rodadas de comunicação de tamanho $O\left(\frac{t^2+kt}{q}\right)$ por processador e tempo/espaço de execução $O\left(\frac{t^2+kt}{q}\right)$.

Em resumo, lembrando que estamos considerando a união de dois GDAGs de dimensão $l\times k$ e que $t=l+k-1$, temos o seguinte teorema:

Teorema 3.11 *Se um GDAG de dimensão $(2l-1)\times k$ (ou $l\times(2k-1)$) tem duas metades de dimensão $l\times k$, e a matriz AL de cada metade está distribuída num conjunto distinto de q processadores, então é possível calcular a matriz AL do GDAG com os $2q$ processadores usando seis rodadas de comunicação de $O\left(\frac{(l+k)^2}{q}+q^3+q(l+k)\right)$ dados por processador e tempo/espaço $O\left(\frac{(l+k)^2}{q}\right)$ por processador.*

3.4.8 Análise do Algoritmo CGM para o ATS

Voltamos então para o problema inicial: resolver o ATS para duas cadeias A e B, de comprimentos respectivamente n_a e n_b. O algoritmo lida com as cadeias de maneira indistinta, encontrando tanto os alinhamentos de A com subcadeias de B quanto os de B com subcadeias de A, por isso iremos supor, sem perda de generalidade, que $n_b \geq n_a$.

Como já mencionado na Seção 3.2, o algoritmo paralelo para o ATS envolve uma etapa inicial em que cada processador calcula a matriz AL para um GDAG, seguida de $\log p$ etapas em que pares de GDAGs são unidos em paralelo. Para estas etapas temos o seguinte lema.

Lema 3.12 *Sendo $p < \sqrt{n_a}$, o tempo e o espaço necessários para realização de uma etapa de união de GDAGs são $O\left(\frac{n_b^2}{p}\right)$. Cada etapa requer um número constante de rodadas de comunicação em que cada processador recebe/envia $O\left(\frac{n_b^2}{p}\right)$ dados.*

Prova. O número de processadores que compartilham a matriz AL de um GDAG no ínicio de uma etapa, q, dobra a cada etapa. As dimensões dos GDAGs dobram a cada *duas* etapas de união. No início da primeira etapa temos $q = 1$ e GDAGs de dimensão $\frac{n_a}{\sqrt{p}} \times \frac{n_b}{\sqrt{p}}$. No Teorema 3.11 a complexidade do processo de união paralela é dada em função da soma das dimensões dos GDAGs, $(l + k)$. Podemos concluir que esta soma no início de cada etapa vale $O\left(\frac{n_b\sqrt{q}}{\sqrt{p}}\right)$.

Supondo que $p < \sqrt{n_a}$ teremos $q < \sqrt{l+k}$ em cada etapa e assim podemos simplificar o resultado do Teorema 3.11, indicando que o tempo, o espaço e a quantidade de dados enviados/recebidos por cada processador em cada rodada de comunicação são todos $O\left(\frac{(l+k)^2}{q}\right) = O\left(\frac{n_b^2}{p}\right)$. □

Finalmente chegamos ao principal resultado deste capítulo:

Teorema 3.13 *O problema ATS para cadeias de comprimento n_a e n_b pode ser resolvido por p processadores no modelo CGM em espaço $O\left(\frac{n_b^2}{p}\right)$, tempo $O\left(\frac{n_b^2}{p}\log n_a\right)$ e $O(\log p)$ rodadas de comunicação, sendo $p^2 < n_a \leq n_b$.*

Prova. Na etapa inicial, cada processador lida com um GDAG de dimensões $\frac{n_a}{\sqrt{p}} \times \frac{n_b}{\sqrt{p}}$, calculando a matriz AL deste GDAG em tempo $O\left(\frac{n_a n_b}{p}\log\frac{n_a}{\sqrt{p}}\right)$, segundo o Teorema 3.5.

A seguir temos as $\log p$ etapas de união paralela de GDAGs, que segundo o Lema 3.12 levarão ao todo tempo $O\left(\frac{n_b^2}{p}\log p\right) = O\left(\frac{n_b^2}{p}\log n_a\right)$. As demais afirmações deste teorema foram demonstradas ao longo desta seção. □

3.5 Comentários

O procedimento de escalonamento dinâmico dos subproblemas para os processadores é pouco usual e envolve detalhes de implementação que precisam ser considerados com cautela. Um destes detalhes é a avaliação dos custos dos subproblemas, que precisa ser feita com alguma precisão para que a distribuição de carga de trabalho seja equilibrada.

Naturalmente, o tempo de execução de qualquer algoritmo em uma máquina *real* não pode ser previsto com precisão absoluta, mas uma estimativa razoável pode ser obtida através do uso de um *modelo* adequado. Para garantir os resultados presentes nesta tese, uma estimativa razoável é suficiente.

Por simplicidade, o tempo de execução de cada subproblema foi calculado com base na duração de cada *varredura* executada, e esta duração foi estimada como sendo proporcional ao número de acessos feitos às matrizes AL. Uma estimativa mais precisa seria dependente do sistema em que o programa fosse executado, podendo ser feita com base em alguns resultados empíricos. As funções v_d e v_o (Definição 3.3, página 41) teriam que ser modificadas com base nestes resultados. No entanto, as seguintes características de v_d e v_o devem se manter:

- Dados os argumentos n, t e k, os valores de $v_d(n,t,k)$ e $v_o(n,t,k)$ devem ser calculáveis em tempo $O(1)$.
- A distribuição feita com base nestas estimativas deve ser boa o suficiente para garantir que o procedimento paralelo de união de GDAGs apresente aceleração linear.

Isto pode ser conseguido com uma pequena variação nas considerações sobre o tempo de execução de cada varredura: dados r (número de trechos a serem considerados na varredura) e k (número de vértices de $meio_U$), o tempo de uma varredura é proporcional a $k + \alpha r + \beta$, para valores de α e β a serem determinados empiricamente (ou analiticamente, usando um modelo adequado na análise do Algoritmo 3.1). Neste capítulo, consideramos $\alpha = 1$ e $\beta = 0$. Mudando o valor de α e β teríamos uma mudança no fator de aproximação (3,621 segundo a equação 3.5, na página 53), mas a aceleração obtida ainda seria linear. Os cálculos desenvolvidos no Capítulo 3 e no Apêndice podem ser adaptados sem problemas.

Capítulo 4

Problema da Maior Subseqüência Comum

Neste capítulo, abordaremos o problema da Maior Subseqüência Comum, mais conhecido pelo seu nome em inglês, *Longest Common Subsequence*, de onde tiramos a sigla LCS (ver Definição 1.2). Parte dos resultados aqui presentes estão em artigo aceito para publicação em anais do *17th International Parallel and Distributed Processing Symposium (IEEE-IPDPS 2003)* [5].

O principal resultado deste capítulo é a apresentação de um algoritmo que resolve o LCS com aceleração linear e $O(\log p)$ rodadas de comunicação. O número inferior de rodadas de comunicação é a principal vantagem deste algoritmo sobre os que podem ser desenvolvidos com base nas idéias do Capítulo 2.

Na verdade, o algoritmo resolve o problema ALCS (Definição 1.4, página 8), mais geral, apresentando o comprimento das maiores subseqüências comuns entre A e *todas* as subcadeias de B. Este resultado é semelhante ao que é obtido através do algoritmo apresentado no Capítulo 3 para o problema ATS. Os usos para estes resultados expandidos são os já apresentados na Seção 1.3.1.

Novamente, iremos nos concentrar primeiramente na determinação dos *comprimentos* das maiores subseqüências comuns. Mais exatamente, considerando o GDAG do problema ALCS queremos encontrar os pesos totais dos melhores caminhos para todos os pares de vértices em que o primeiro está na linha superior do GDAG e o segundo na linha inferior. A solução para este problema pode ser escrita em uma matriz, como foi feito no Capítulo 3 para o problema ATS.

Uma vez exposto o algoritmo que determina os comprimentos das maiores subseqüências comuns, veremos que as estruturas remanescentes da execução do algoritmo permitem a recuperação eficiente da maior subseqüência comum entre A e B.

Apesar da semelhança entre este problema e o ATS, é possível resolvê-lo seqüencialmente em tempo $O(n_a n_b)$, melhor do que o tempo para o ATS por um fator logarítmico. Assim, apesar de o problema ALCS ser mais complexo do que o LCS, o tempo para sua solução é equivalente

ao do LCS a menos de uma constante multiplicativa. O mesmo não acontece com o problema ATS e sua versão mais simples de alinhamento de cadeias. Estas informações se encontram sumarizadas na Tabela 1.1 (página 8).

Há duas razões para nos concentrarmos no problema ALCS e não no LCS básico. Em primeiro lugar, temos mais informações como resultado, sem grande prejuízo no tempo de execução. Em segundo lugar, o ALCS é mais adequado do que o LCS para uma paralelização do tipo Divisão-e-Conquista.

O GDAG do problema ALCS é mais simples do que o GDAG do problema ATS. Em particular, o primeiro só apresenta arcos de peso 0 ou 1. Esta diferença permite o surgimento de propriedades interessantes que serão comentadas na próxima seção. A partir destas propriedades pode-se desenvolver um algoritmo seqüencial para o ALCS, apresentado na Seção 4.2. Segue-se uma descrição do algoritmo paralelo CGM, o principal resultado deste capítulo.

O algoritmo aqui exposto apresenta aceleração linear com relação ao tempo de pior caso dos algoritmos de uso corrente. É o primeiro algoritmo, pelo nosso conhecimento, que resolve o problema LCS com aceleração linear e apenas $O(\log p)$ rodadas de comunicação no modelo CGM. Cada rodada de comunicação envolve o envio/recebimento de pouco mais do que $O(n_a + n_b)$ dados por processador. O espaço utilizado é também reduzido, graças a uma estrutura de dados inédita especialmente desenvolvida para o problema, exposta na Seção 4.3.

4.1 Propriedades do Problema ALCS

Consideremos o GDAG G correspondente ao LCS para duas cadeias A e B de comprimento n_a e n_b, respectivamente. Este GDAG possui $n_a + 1$ linhas e $n_b + 1$ colunas de vértices. Todos os arcos verticais e horizontais tem peso 0. Numerando as linhas e colunas a partir de 0 (ver Figura 4.1), o arco entre o vértice na posição $(i-1, j-1)$ e o vértice na posição (i, j) tem peso 1 se $a_i = b_j$. Se $a_i \neq b_j$, este arco tem peso 0, podendo ser desprezado

Os vértices da linha superior (topo) de G serão denotados por $T_G(i)$, e os da linha inferior (fundo) de G por $F_G(i)$, $0 \leq i \leq n_b$. Temos então a seguinte definição:

Definição 4.1 ($C_G(i,j)$) *Para $0 \leq i \leq j \leq n_b$, $C_G(i,j)$ é o peso total do maior caminho entre os vértices $T_G(i)$ e $F_G(j)$. Caso $i > j$ (não existe caminho entre $T_G(i)$ e $F_G(j)$), $C_G(i,j) = 0$.*

$C_G(i,j)$ representa o comprimento da maior subseqüência comum entre A e a subcadeia B_{i+1}^{j}. Quando i = j, a subcadeia não existe e o único caminho entre $T_G(i)$ e $F_G(j)$ tem peso 0 (é formado apenas por arcos verticais). Quando $i > j$, B_{i+1}^{j} também não existe e adota-se $C_G(i,j) = 0$.

A seguir são mostradas algumas propriedades dos valores de $C_G(i,j)$.

Propriedades 4.1

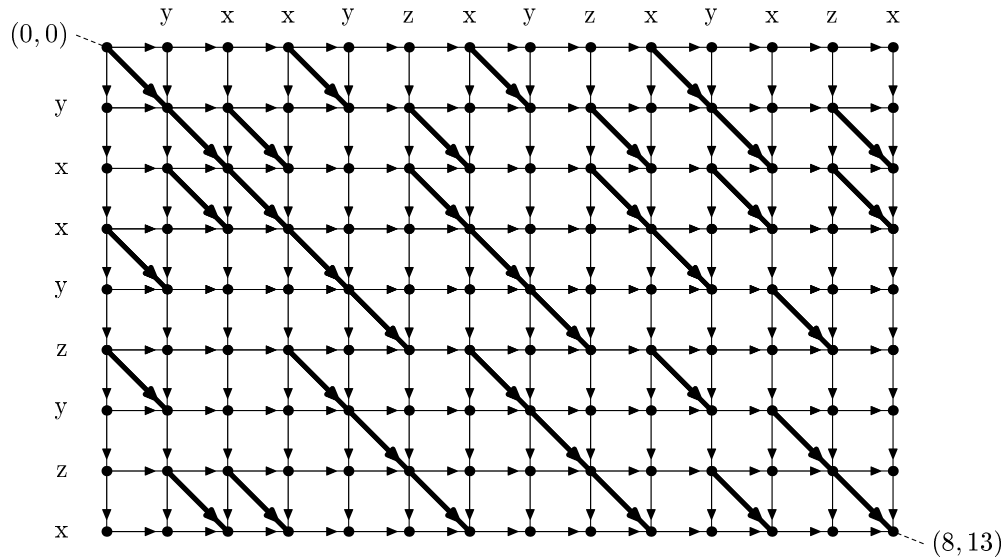

Figura 4.1: GDAG para o problema ALCS. As cadeias comparadas são A = yxxyzyzx e B = yxxyzxyzxyxzx. Os arcos grossos nas diagonais têm peso 1, os demais têm peso 0.

1. *Para todo i ($0 \leq i \leq n_b$) e todo j ($1 \leq j \leq n_b$) temos $C_G(i,j) = C_G(i,j-1)$ ou $C_G(i,j) = C_G(i,j-1)+1$.*

2. *Para todo i ($1 \leq i \leq n_b$) e todo j ($0 \leq j \leq n_b$) temos $C_G(i-1,j) = C_G(i,j)$ ou $C_G(i-1,j) = C_G(i,j)+1$.*

3. *Para todo i e todo j ($1 \leq i < j \leq n_b$) temos que se $C_G(i-1,j) = C_G(i-1,j-1)+1$ então $C_G(i,j) = C_G(i,j-1)+1$.*

Prova. Demonstraremos a propriedade 4.1(1). A demonstração de 4.1(2) é análoga.

Naturalmente, $C_G(i,j)$ e $C_G(i,j-1)$ são inteiros. Quando $i \geq j$ temos $C_G(i,j) = C_G(i,j-1) = 0$. Para o caso $i < j$, analisando os caminhos no GDAG a partir do vértice $T_G(i)$ fica claro que $C_G(i,j) \geq C_G(i,j-1)$, pois o melhor caminho até $F_G(j-1)$ pode ser estendido até $F_G(j)$ por um arco horizontal de peso 0. Por outro lado, como todos os arcos de peso 1 são diagonais, fica claro que $C_G(i,j) - C_G(i,j-1) \leq 1$ pois um caminho de $T_G(i)$ até $F_G(j-1)$ pode ser criado a partir do melhor caminho até $F_G(j)$, eliminando-se os vértices na coluna j e completando-se o caminho com arcos verticais até $F_G(j-1)$. No máximo uma diagonal é eliminada no processo.

Para demonstrar 4.1(3), suponhamos que $C_G(i-1,j) = C_G(i-1,j-1)+1$ para um certo par (i,j), $1 \leq i < j \leq n_b$. Seja $C1$ o caminho entre $T_G(i-1)$ e $F_G(j)$ de peso $C_G(i-1,j)$ tomado de tal forma que não há outro caminho de mesmo peso passando por vértices à esquerda dos vértices de $C1$. Seja $C2$ um caminho entre $T_G(i)$ e $F_G(j-1)$ de peso total $C_G(i,j-1)$, tomado de maneira semelhante à de $C1$. É fácil provar que estes caminhos devem ter um único trecho em comum com pelo menos um vértice. Sejam $C1_a$ e $C1_b$ os trechos de $C1$ não compartilhados com $C2$, sendo $C1_a$ o trecho iniciado em $T_G(i-1)$ e $C1_b$ o trecho terminado em $F_G(j)$. Sejam

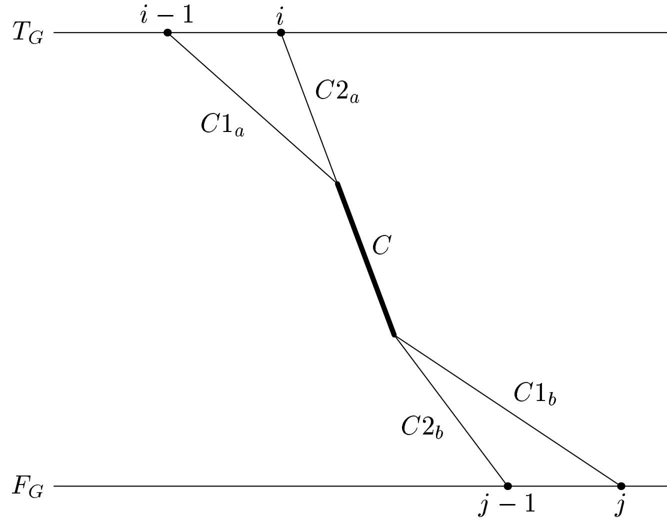

Figura 4.2: Demonstração da Propriedade 4.1(3).

$C2_a$ e $C2_b$ trechos de $C2$ definidos de forma semelhante. O trecho comum será chamado C. Esta nomenclatura está representada na Figura 4.2.

Naturalmente, o peso de $C1_b$ deve ser igual ao peso de $C2_b$ acrescido de 1, do contrário teríamos um caminho entre $T_G(i-1)$ e $F_G(j-1)$ (formado por $C1_a$, C e $C2_b$) de mesmo peso que $C1$, o que contradiz a hipótese original. Assim, o caminho formado por $C2_a$, C e $C1_b$ terá peso igual ao de $C2$ acrescido de 1, o que implica $C_G(i,j) > C_G(i,j-1)$. De 4.1(1) concluímos que $C_G(i,j) = C_G(i,j-1)+1$.

Deve-se notar que esta demonstração depende da existência dos caminhos indicados, que é garantida pelas desigualdades $1 \leq i < j \leq n_b$. □

A Propriedade 4.1(3) é uma variação das *propriedades de Monge*. Em [40] este tipo de propriedade é extensamente usada para a resolução do problema ATS, em particular para casos que se assemelham ao ALCS. Estes resultados serão comentados mais à frente, na exposição do algoritmo seqüencial para o ALCS.

A Propriedade 4.1(1) é importante por indicar que, para um dado i fixo, os valores $C_G(i,0)$, $C_G(i,1)$, $C_G(i,2)$, $\ldots C_G(i,n_b)$ formam uma seqüência não decrescente que pode ser dada de maneira implícita, indicando apenas os valores de j para os quais $C_G(i,j) > C_G(i,j-1)$.

Este fato foi usado em vários algoritmos seqüenciais para o LCS, como em [8, 28, 39]. Também é importante para o algoritmo CREW-PRAM apresentado em [32], base para o algoritmo desenvolvido neste capítulo e para a definição a seguir:

Definição 4.2 (Matriz D_G) *Seja G o GDAG para o problema LCS entre as cadeias A e B. Para $0 \leq i \leq n_b$, $D_G(i,0) = i$ e para $1 \leq k \leq n_a$, $D_G(i,k)$ indica o valor de j tal que $C_G(i,j) = k$ e $C_G(i,j-1) = k-1$. Se não houver tal valor então $D_G(i,k) = \infty$.*

Implícito nesta definição está o fato de que $C(i,j) \leq n_a$. Definimos $D_G(i,0) = i$ e não 0 porque isto simplifica a exposição do algoritmo paralelo, como ficará claro mais à frente. Note que, por conveniência de notação, D_G é uma matriz com índices iniciados em 0. Denotaremos D_G^i a linha de índice i de D_G, ou seja, o vetor linha formado por $D_G(i,0)$, $D_G(i,1)$, ..., $D_G(i,n_a)$. A matriz D_G é essencial ao algoritmo CGM que será apresentado, sendo a principal forma de representação usada durante a execução do algoritmo.

Como exemplo, tomemos o GDAG da Figura 4.1. Os valores de $C_G(i,j)$ e $D_G(i,k)$ são mostrados nas tabelas 4.1 e 4.2 (página 62).

Através dos valores de $D_G(i,k)$ pode-se obter os valores de $C_G(i,k)$. Além disso, os resultados a seguir [31, 32] indicam que a informação contida nos valores de $D_G(i,k)$ pode ser representada em espaço $O(n_a + n_b)$, ao invés do espaço $O(n_a n_b)$ necessário para a representação direta.

Propriedades 4.2

1. *Se $k_1 < k_2$ e $D_G(i,k_1) \neq \infty$ então $D_G(i,k_1) < D_G(i,k_2)$.*

2. *Se $i_1 < i_2$ então $D_G(i_1,k) \leq D_G(i_2,k)$.*

3. *Se $D_G(i_1,k_1) = j_1$ e $j_1 \neq \infty$ então para todo i, $i_1 \leq i \leq j_1$ existe um k tal que $D_G(i,k) = j_1$. Em outras palavras, se o componente j_1 aparece na linha D_G^i então ele irá aparecer em todas as linhas até a linha $D_G^{j_1}$.*

4. *Se $D_G(i_1,k_1) = D_G(i_2,k_2) = j_1$ então para todo i, $i_1 \leq i \leq i_2$ existe um k tal que $D_G(i,k) = j_1$. Em outras palavras, se o componente j_1 aparece nas linhas $D_G^{i_1}$ e $D_G^{i_2}$ então ele irá aparecer em todas as linhas entre $D_G^{i_1}$ e $D_G^{i_2}$.*

Prova. Se $k_1 < k_2$ e $j_1 = D_G(i,k_1) \neq \infty$, $C_G(i,j_1) = k_1$. $C_G(i,j)$ não decresce com j, logo se existir j tal que $C_G(i,j) = k_2$ então $j > j_1$. Se não existir tal valor então $D_G(i,k_2) = \infty$. Em qualquer um dos dois casos teremos $D_G(i,k_1) < D_G(i,k_2)$, o que demonstra (1).

Se $i_1 < i_2$, pela Propriedade 4.1(2) $C_G(i_2,j) \leq C_G(i_1,j)$ para todo j. Se $D_G(i_1,k) > D_G(i_2,k) \neq \infty$ para algum k, então para $j = D_G(i_2,k)$ teríamos $C_G(i_2,j) = k > C_G(i_1,j)$, uma contradição. Isto demonstra (2).

A demonstração de (3) é feita com indução sobre i. Suponhamos que para um certo valor de i ($i_1 \leq i < j_1 \neq \infty$) temos $D_G(i,k) = j_1$ para algum valor de k (não importa qual). Isto significa que $C_G(i,j_1) = k = C_G(i,j_1 - 1) + 1$. Pela Propriedade 4.1(3) temos que $C_G(i+1,j_1) = C_G(i+1,j_1-1)+1$, ou seja, existe algum k tal que $D_G(i+1,k) = j_1$. Este passo indutivo pode ser usado enquanto a condição para a Propriedade 4.1(3) ($i + 1 < j_1$) é garantida. Assim, j_1 aparece em todas as linhas de $D_G^{i_1}$ até $D_G^{j_1-1}$. A presença de j_1 em $D_G^{j_1}$ decorre da Definição 4.2.

A demonstração de (4) é imediata a partir de (3), uma vez que $D_G(i,k)$ só pode ser j_1 se $i \leq j_1$. □

As propriedades 4.2(2) e 4.2(3) são usadas para reduzir o espaço necessário para representação de D_G, pois indicam que duas linhas consecutivas de D_G são muito semelhantes. Mais exatamente, podemos determinar as seguintes propriedades:

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

Tabela 4.1: C_G referente ao GDAG da Figura 4.1.

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

Tabela 4.2: D_G referente ao GDAG da Figura 4.1.

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

Tabela 4.3: V_G referente ao GDAG da Figura 4.1. Complementa D_G^0 (primeira linha da Tabela 4.2) na representação de uma solução para o ALCS.

Propriedades 4.3 *Para* $0 \leq i < n_b$,

1. *Existe um único componente finito de* D_G^i *que não aparece em* D_G^{i+1}, *que é* $D_G(i,0) = i$.
2. *Existe no máximo um componente finito de* D_G^{i+1} *que não aparece em* D_G^i.

Prova. A afirmação (1) decorre diretamente da Propriedade 4.2(3). Se j_1 aparece na linha D_G^i então ele irá aparecer em todas as linhas seguintes, até a linha j_1. Como $D_G(i,0) = i$, esta é a última linha em que i aparece. Todos os demais componentes de D_G^i são maiores do que i (pela Propriedade 4.2(1)) e irão aparecer em D_G^{i+1}.

Pela Propriedade 4.2(2) temos que o número de componentes finitos de D_G^{i+1} não pode ser maior do que o de D_G^i. Pelo que foi exposto anteriormente, sabemos que há um único componente finito de D_G^i que não aparece em D_G^{i+1}, portanto D_G^{i+1} pode conter no máximo um "novo" componente. □

A Propriedades 4.3(1) e 4.3(2) indicam que podemos transformar uma linha de D_G em sua linha seguinte através da remoção de um componente (o primeiro) e inserção de um novo componente (finito ou não). Dizemos então que duas linhas consecutivas de D_G são *1-variantes*. Se tomarmos $r+1$ linhas consecutivas teremos que elas são r*-variantes*, pois de qualquer uma destas linhas podemos obter a outra através da remoção e inserção de no máximo r componentes.

Isto sugere uma representação em espaço $O(n_a + n_b)$ para D_G. Esta representação é composta por D_G^0 (a primeira linha de D_G), que ocupa espaço $O(n_a)$, e pelo vetor V_G, de tamanho $O(n_b)$, descrito a seguir:

Definição 4.3 (Vetor V_G) *Para* $1 \leq i \leq n_b$, $V_G(i)$ *é o valor do componente finito que está presente na linha* D_G^i *mas não na linha* D_G^{i-1}. *Se não houver tal componente finito,* $V_G(i) = \infty$.

A Tabela 4.3 (página 62) mostra V_G para o GDAG da Figura 4.1. É interessante comparar as três representações já comentadas: por C_G (Tabela 4.1), por D_G (Tabela 4.2) e esta última.

A estrutura composta por V_G e D_G^0 será importante no algoritmo paralelo, pois a estratégia a ser usada envolve a divisão do GDAG original em faixas e a resolução do ALCS em cada faixa. Estas faixas são então unidas, o que requer o envio da resposta dos ALCS parciais entre os processadores. A forma compacta das respostas parciais torna mais curtas as etapas de comunicação.

A seguir será dado um algoritmo seqüencial para o ALCS.

4.2 Algoritmo Seqüencial para o ALCS

O algoritmo apresentado nesta seção é uma adaptação de um algoritmo apresentado em [40] para solução do problema ATS quando os custos de edição são inteiros. Esta apresentação será mais completa e mais específica para o problema ALCS. Várias propriedades adicionais são apresentadas, sendo que estas propriedades são usadas apenas nesta seção. Em outras palavras,

a leitura das seções posteriores pode ser feita de maneira independente da leitura desta seção, desde que se conheça o seu resultado básico: para cadeias de comprimento n_a e n_b pode-se resolver o ALCS seqüencialmente em tempo $O(n_a n_b)$ e espaço $O(n_a + n_b)$ (ou $O(n_a n_b)$, se além dos comprimentos também quisermos obter as subseqüências).

4.2.1 Propriedades do ALCS Usadas no Algoritmo Seqüencial

Para cada vértice do GDAG G do problema é preciso determinar alguma informação sobre a distância entre este vértice e cada vértice da linha superior de G. Para isso adotamos a seguinte definição:

Definição 4.4 ($C_G^l(i,j)$) *Para $0 \leq l \leq n_a$, $0 \leq i \leq n_b$ e $0 \leq j \leq n_b$ $C_G^l(i,j)$ é o peso total do maior caminho entre os vértices $(0,i)$ e (l,j), se tal caminho existir. Caso não exista um caminho (o que ocorre se $i > j$), $C_G^l(i,j) = 0$.*

Esta definição é uma extensão da Definição 4.1 para lidar com os vértices internos do GDAG G (note que $C_G(i,j) = C_G^{n_a}(i,j)$). Com isto podemos citar as seguintes propriedades, que lidam com vértices vizinhos em G:

Propriedades 4.4 *Para todo l $(0 \leq l \leq n_a)$ temos:*

1. *Para todo i $(0 \leq i \leq n_b)$ e todo j $(1 \leq j \leq n_b)$ temos $C_G^l(i,j) = C_G^l(i,j-1)$ ou $C_G^l(i,j) = C_G^l(i,j-1) + 1$.*
2. *Para todo i e todo j $(0 < i < j \leq n_b)$ temos que se $C_G^l(i-1,j) = C_G^l(i-1,j-1) + 1$ então $C_G^l(i,j) = C_G^l(i,j-1) + 1$.*

A demonstração destas propriedades é omitida, por ser idêntica à já apresentada para a Propriedade 4.1. A inclusão de l nestas propriedades não altera a natureza delas.

As Propriedades 4.4 se referem às diferenças de distâncias para dois vértices internos de G que são vizinhos em uma mesma *linha*. As propriedades abaixo se referem às diferenças para dois vértices internos vizinhos em uma mesma *coluna*.

Propriedades 4.5 *Para todo l $(1 \leq l \leq n_a)$ temos:*

1. *Para todo i $(0 \leq i \leq n_b)$ e todo j $(0 \leq j \leq n_b)$ temos $C_G^l(i,j) = C_G^{l-1}(i,j)$ ou $C_G^l(i,j) = C_G^{l-1}(i,j) + 1$.*
2. *Para todo i e todo j $(0 < i \leq j \leq n_b)$ temos que se $C_G^l(i-1,j) = C_G^{l-1}(i-1,j)$ então $C_G^l(i,j) = C_G^{l-1}(i,j)$.*

As demonstrações das propriedades anteriores são semelhantes às já apresentadas para a Propriedade 4.1, sendo assim omitidas.

Uma conseqüência das Propriedades 4.4 e 4.5 é que é possível codificar *variações* nos caminhos para os vértices internos do GDAG de maneira eficiente. Se tomarmos um vértice na linha superior do GDAG, digamos $T_G(i)$, e a partir dele traçarmos os maiores caminhos possíveis para dois vértices vizinhos numa mesma linha, $(l, j-1)$ e (l, j), estes caminhos terão o mesmo peso se i estiver abaixo de um certo limite. Para valores de i iguais ou superiores a este limite, o peso do caminho até (l, j) será superior (por 1) ao do caminho até $(l, j-1)$. O valor deste limite depende de l e j e será denotado por $i_h(l, j)$.

De modo semelhante, os pesos dos maiores caminhos a partir de $T_G(i)$ para vértices vizinhos numa mesma coluna, $(l-1, j)$ e (l, j), diferem por 1 se i estiver *abaixo* de um certo limite $i_v(l, j)$. No limite ou acima dele, os caminhos têm o mesmo peso.

Mais formalmente, temos as seguintes definições:

Definição 4.5 **($i_h(l,j)$)** *Para todo (l, j), $0 \leq l \leq n_a$ e $1 \leq j \leq n_b$, $i_h(l, j)$ é o menor valor de $i < j$ tal que $C^l_G(i, j) = C^l_G(i, j-1) + 1$. Caso tal i não exista, $i_h(l, j) = j$.*

Definição 4.6 **($i_v(l,j)$)** *Para todo (l, j), $1 \leq l \leq n_a$ e $1 \leq j \leq n_b$, $i_v(l, j)$ é o menor valor de $i \leq j$ tal que $C^l_G(i, j) = C^{l-1}_G(i, j)$.*

O algoritmo para o ALCS se baseia na determinação dos limites i_h e i_v para todos os vértices do GDAG. Antes de prosseguirmos, uma observação a respeito de i_h: a existência deste limite já foi esboçada na Propriedade 4.2(3). O primeiro valor de i que faz com que exista um k tal que $D_G(i, k) = j$ é $i_h(n_a, j)$. Para valores de i acima deste limite, j continua aparecendo em D^i_G, o que significa que há uma diferença nos maiores caminhos de $T_G(i)$ para $(n_a, j-1)$ e (n_a, j).

Além de i_h e i_v, para cada vértice precisaremos determinar mais dois limites, i_1 e i_2, explicados a seguir. Dado um vértice (l, j), sendo $0 < l \leq n_a$ e $0 < j \leq n_b$, os melhores caminhos do topo do GDAG até ele devem ter como penúltimo vértice $(l, j-1)$, $(l-1, j-1)$ ou $(l-1, j)$. Procurando sempre os caminhos mais à esquerda, veremos que eles obedecem à seguinte propriedade:

Propriedade 4.6 *Para todo (l, j), $1 \leq l \leq n_a$ e $0 < j \leq n_b$, existem valores i_1 e i_2 tais que o melhor caminho (mais à esquerda) de $T_G(i)$ para (l, j) tem como penúltimo vértice:*

- $(l, j-1)$ *se* $0 \leq i < i_1$,
- $(l-1, j-1)$ *se* $i_1 \leq i < i_2$,
- $(l-1, j)$ *se* $i_2 \leq i < j$.

A Figura 4.3 ilustra a Propriedade 4.6.

Esta propriedade é uma variante da Propriedade 3.1. A demonstração é muito semelhante, baseando-se também na impossibilidade de haver cruzamentos entre os melhores caminhos que partem de dois vértices distintos de $T_G(i)$. Os detalhes serão omitidos.

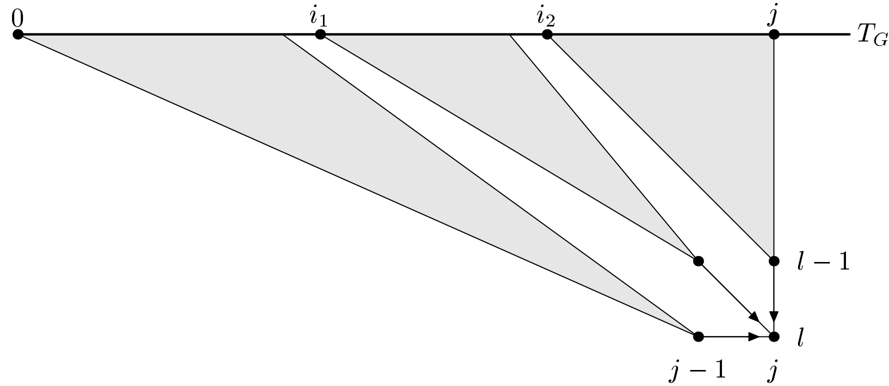

Figura 4.3: Penúltimo vértice de cada caminho em um GDAG. A partir do topo de um GDAG, os caminhos que chegam em um vértice interior passam pelos vértices adjacentes como ilustrado.

4.2.2 Descrição e Análise do Algoritmo Seqüencial

A determinação dos quatro limites citados (i_h, i_v, i_1 e i_2) é feita vértice a vértice, varrendo o GDAG em linhas (ou em colunas). Para calcular os limites de um vértice (l, j) é necessário conhecer apenas $i_h(l-1, j)$ e $i_v(l, j-1)$, pois estes valores indicam diferenças entre os caminhos para os vértices adjacentes e anteriores a (l, j) (vamos chamá-los de vértices *candidatos*). Dois casos distintos devem ser considerados separadamente:

Caso 1 Se $a_l \neq b_j$, ou seja, o arco entre $(l-1, j-1)$ e (l, j) tiver peso 0, este arco pode ser ignorado e o penúltimo vértice de qualquer caminho até (l, j) não pode ser $(l-1, j-1)$. Neste caso, $i_1 = i_2$. Há duas possibilidades a partir deste ponto:

Caso 1.i $i_v(l, j-1) \leq i_h(l-1, j)$: há três faixas de valores a considerar para i na escolha pelo penúltimo vértice no caminho para (l, j):

- para $0 \leq i < i_v(l, j-1)$ o melhor caminho de $T_G(i)$ para $(l, j-1)$ é melhor do que o melhor caminho para os outros dois candidatos, logo o vértice escolhido é $(l, j-1)$.
- $i_v(l, j-1) \leq i < i_h(l-1, j)$: os melhores caminhos de $T_G(i)$ para cada um dos três candidatos têm o mesmo peso, logo o vértice escolhido é $(l, j-1)$ novamente (por estar mais à esquerda).
- $i_h(l-1, j) \leq i \leq j$: o melhor caminho de $T_G(i)$ para $(l-1, j)$ é melhor do que o melhor caminho para os outros dois candidatos. O vértice escolhido é $(l-1, j)$.

Assim, $i_1 = i_2 = i_h(l-1, j)$, $i_h(l, j) = i_h(l-1, j)$ e $i_v(l, j) = i_v(l, j-1)$.

Caso 1.ii $i_v(l, j-1) > i_h(l-1, j)$: há novamente três faixas de valores a considerar para i na escolha pelo penúltimo vértice no caminho para (l, j):

- para $0 \leq i < i_h(l-1, j)$: o vértice escolhido é $(l, j-1)$.

- $i_h(l-1,j) \leq i < i_v(l,j-1)$: os melhores caminhos de $T_G(i)$ para $(l,j-1)$ e para $(l-1,j)$ têm o mesmo peso (o caminho para $(l-1,j-1)$ é menor). Novamente, o vértice escolhido é $(l,j-1)$ por estar mais à esquerda.
- $i_v(l,j-1) \leq i \leq j$: o vértice escolhido é $(l-1,j)$.

Temos então $i_1 = i_2 = i_v(l,j-1)$, $i_h(l,j) = i_v(l,j-1)$ e $i_v(l,j) = i_h(l-1,j)$.

Em resumo, no Caso 1 temos $i_1 = i_2 = i_h(l,j) = \max(i_v(l,j-1), i_h(l-1,j))$ e $i_v(l,j) = \min(i_v(l,j-1), i_h(l-1,j))$.

Caso 2 Se $a_l = b_j$, ou seja, o arco entre $(l-1,j-1)$ e (l,j) tiver peso 1, temos que considerar os três candidatos, sendo que $(l-1,j-1)$ tem a "vantagem" de estar ligado a um arco de peso 1. Na verdade, como procuramos sempre pelo caminho mais à esquerda, praticamente nenhum caminho irá passar por $(l-1,j)$. A exceção ocorre quando $i = j$, pois não há caminho de $T_G(i)$ para $(l-1,j-1)$, o que implica $i_2 = j$.

Quando $i < i_v(l,j-1)$ o melhor caminho de $T_G(i)$ para $(l,j-1)$ tem peso superior ao do caminho para $(l-1,j-1)$, compensando o peso 1 do arco entre $(l-1,j-1)$ e (l,j). Assim, $i_1 = i_v(l,j-1)$ e $i_h(l,j) = i_v(l,j-1)$. Por razões semelhantes, temos $i_v(l,j) = i_h(l-1,j)$.

Os cálculos acima se aplicam a vértices não pertencentes às bordas superior e esquerda do GDAG. Para os vértices da borda superior fazemos $i_h(0,j) = j$ $(1 \leq j \leq n_b)$ e para os vértices da borda esquerda fazemos $i_v(l,0) = 0$ $(1 \leq l \leq n_a)$. Os outros valores não são usados.

A partir dos limites $i_h(n_a,j)$ com $1 \leq j < n_b$ (relativos aos vértices da borda inferior do GDAG) é possível determinar a solução para o ALCS na codificação que está sendo usada neste capítulo (usando D_G^0 e V_G). Primeiramente, faz-se $D_G^0(0) = 0$. Depois, se para um certo j temos $i_h(n_a,j) = 0$, isto significa que j é componente de D_G^0. Se $i_h(n_a,j) = i, 1 \leq i \leq j$, isto significa que a primeira linha de D_G em que j aparece é D_G^i, logo $V_G(i) = j$.

O Algoritmo 4.1 torna explícito este procedimento. Note que o uso dos limites i_1 e i_2 não é necessário para o cálculo de D_G^0 e V_G, portanto estes limites não foram incluídos no algoritmo. No entanto, se for necessária a recuperação rápida do melhor caminho entre dois vértices $T_G(i)$ e $F_G(j)$ quaisquer, o cálculo e o armazenamento dos limites i_1 e i_2 para todos os vértices do GDAG são necessários: estes limites permitem a construção do caminho a partir de $F_G(j)$ retrocedendo até $T_G(i)$, em tempo $O(n_a + n_b)$.

Algoritmo 4.1: ALCS seqüencial.
Entrada: Cadeias $A = a_1 a_2 \ldots a_{n_a}$ e $B = b_1 b_2 \ldots b_{n_b}$.
Saída: Vetores D_G^0 e V_G relativos às cadeias A e B.

1 Para $j \leftarrow 0$ até n_b faça
1.1 $\quad i_h(0,j) \leftarrow j$
2 Para $l \leftarrow 0$ até n_a faça

2.1	$\quad\quad i_v(l, 0) \leftarrow 0$
3	Para $l \leftarrow 1$ até n_a faça
3.1	$\quad$ Para $j \leftarrow 1$ até n_b faça
3.1.1	$\quad\quad$ Se $a_l \neq b_j$ então
3.1.1.1	$\quad\quad\quad i_h(l, j) \leftarrow \max(i_v(l, j-1), i_h(l-1, j))$
3.1.1.2	$\quad\quad\quad i_v(l, j) \leftarrow \min(i_v(l, j-1), i_h(l-1, j))$
3.1.2	$\quad\quad$ Senão
3.1.2.1	$\quad\quad\quad i_h(l, j) \leftarrow i_v(l, j-1)$
3.1.2.2	$\quad\quad\quad i_v(l, j) \leftarrow i_h(l-1, j)$
4	Para $j \leftarrow 1$ até n_b faça
4.1	$\quad V_G(j) \leftarrow \infty$
5	$D_G^0(0) \leftarrow 0$
6	$i \leftarrow 1$
7	Para $j \leftarrow 1$ até n_b faça
7.1	$\quad$ Se $i_h(n_a, j) = 0$ então
7.1.1	$\quad\quad D_G^0(i) \leftarrow j$
7.1.2	$\quad\quad i \leftarrow i + 1$
7.2	$\quad$ Senão
7.2.1	$\quad\quad V_G(i_h(n_a, j)) \leftarrow j$
8	Para $l \leftarrow i$ até n_a faça
8.1	$\quad D_G^0(l) \leftarrow \infty$

fim do algoritmo.

Um exemplo dos resultados deste algoritmo pode ser visto na Tabela 4.4, onde os valores de i_h e i_v são mostrados para cada vértice do GDAG da Figura 4.1 (página 59). Os resultados na última linha da tabela (referente a $i_h(n_a, j)$) levam corretamente aos resultados mostrados nas Tabelas 4.2 e 4.3 (página 62).

Teorema 4.1 *Dadas duas cadeias A e B, de comprimento n_a e n_b respectivamente, é possível resolver (seqüencialmente) o problema ALCS para A e B em tempo $O(n_a n_b)$ e espaço $O(n_b)$ (ou $O(n_a n_b)$ se a recuperação posterior dos melhores caminhos for necessária).*

Prova. O tempo de execução é claramente definido pelos laços aninhados nas linhas 3 e 3.1, sendo $O(n_a n_b)$. No caso em que apenas as distâncias são importantes, pode-se desprezar os valores de $i_h(l, j)$ após a linha $l + 1$ ser processada. Assim, para utilizar os valores $i_h(l, j)$ é necessário espaço $O(n_b)$. O valor de $i_v(l, j)$ só precisa ser mantido para o cálculo de $i_v(l, j + 1)$, logo o espaço necessário para os valores de $i_v(l, j)$ é $O(1)$.

Para armazenar as respostas nos vetores D_G^0 (espaço $O(n_a)$) e V_G (espaço $O(n_b)$) seria necessário espaço $O(n_a + n_b)$. Na verdade, D_G^0 pode ser armazenado em espaço $O(n_b)$, pois se

		j b_j													
		0	1	2	3	4	5	6	7	8	9	10	11	12	13
			y	x	x	y	z	x	y	z	x	y	x	z	x
l a_l	0	—	—	—	—	—	—	—	—	—	—	—	—	—	—
		—	1	2	3	4	5	6	7	8	9	10	11	12	13
	1 y	0	1	1	1	4	4	4	7	7	7	10	10	10	10
		—	0	2	3	1	5	6	4	8	9	7	11	12	13
	2 x	0	0	2	3	1	1	6	4	4	9	7	11	11	13
		—	0	0	2	3	5	1	6	8	4	9	7	12	11
	3 x	0	0	0	2	2	2	1	1	1	4	4	7	7	11
		—	0	0	0	3	5	2	6	8	1	9	4	12	7
	4 y	0	0	0	0	3	3	2	6	6	1	9	4	4	4
		—	0	0	0	0	5	3	2	8	6	1	9	12	7
	5 z	0	0	0	0	0	5	3	2	8	6	1	1	12	7
		—	0	0	0	0	0	5	3	2	8	6	9	1	12
	6 y	0	0	0	0	0	0	0	3	2	2	6	6	1	1
		—	0	0	0	0	0	5	0	3	8	2	9	6	12
	7 z	0	0	0	0	0	0	0	0	3	3	2	2	6	6
		—	0	0	0	0	0	5	0	0	8	3	9	2	12
	8 x	0	0	0	0	0	0	5	0	0	8	3	9	2	12
		—	0	0	0	0	0	0	5	0	0	8	3	9	2

Tabela 4.4: Resultados parciais da execução do Algoritmo 4.1 para o GDAG da Figura 4.1. Em cada célula, o número de cima representa $i_v(l,j)$ e o de baixo $i_h(l,j)$.

$n_a > n_b$ todos os valores de $D_G^0(k)$ são infinitos para $k > n_b$. Por simplicidade, consideraremos que $n_a \leq n_b$.

Caso a recuperação dos caminhos seja necessária, o espaço necessário sobe para $O(n_a n_b)$, para armazenamento dos valores de i_1 e i_2. $\square$

Concluímos assim a apresentação sobre o algoritmo seqüencial. Antes da apresentação do algoritmo CGM será mostrada uma estrutura de dados que permite consultas rápidas aos componentes de D_G e tem tamanho inferior ao de D_G.

4.3 Estrutura Compacta para Armazenamento de D_G

Até o momento estamos considerando duas possíveis representações para a matriz D_G. A representação "direta" apresenta muita redundância e portanto ocupa muito espaço ($O(n_a n_b)$), mas dados os valores de i e k a obtenção de $D_G(i,k)$ pode ser feita em tempo O(1). Já a representação "incremental", feita através dos vetores D_G^0 e V_G, ocupa apenas espaço $O(n_b)$ mas não permite uma consulta rápida aos valores de D_G.

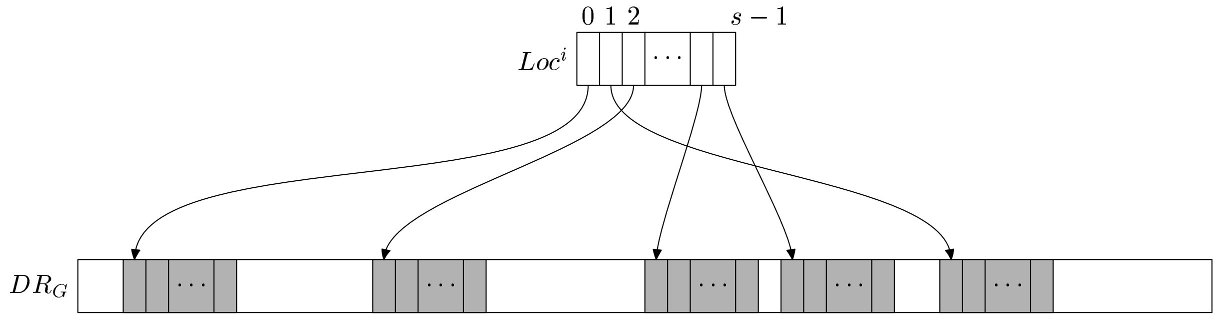

Figura 4.4: Armazenamento dos dados de D_G^i em DR_G.

Nesta seção apresentamos uma estrutura de dados eficiente para o armazenamento de D_G. Esta estrutura ocupa espaço $O\left(\sqrt{n_a}n_b\right)$, permite consultas (apenas para leitura) em tempo $O(1)$ e sua construção pode ser feita com base em D_G^0 e V_G em tempo $O\left(\sqrt{n_a}n_b\right)$. Esta estrutura é essencial para o algoritmo CGM que será apresentado mais à frente.

Um dos principais parâmetros de construção desta estrutura é $\lceil\sqrt{n_a+1}\rceil$, que passaremos a chamar s.

Os dados de D_G são distribuídos em um vetor DR_G (D *reduzido* de G) de tamanho $O(n_b s)$. Consideremos primeiro uma única linha de D_G. D_G^i $(0 \leq i \leq n_b)$ tem tamanho $n_a + 1$. Ela é dividida em s subvetores, todos de tamanho s com a possível exceção do último. Estes vetores são armazenados em locais separados de DR_G, sendo necessário um vetor adicional de tamanho s para indicar onde está cada subvetor. Chamaremos este vetor adicional de Loc^i. A matriz $(n_b+1) \times s$ formada por todos os vetores Loc^i (um para cada D_G^i) será chamada Loc. Os índices de Loc começam em 0. Esta estrutura é mostrada na Figura 4.4.

A estrutura completa inclui o vetor DR_G e a matriz Loc. Antes de prosseguirmos na descrição, provamos o seguinte lema com base no que já foi apresentado:

Lema 4.2 *Através de DR_G e Loc pode-se obter qualquer componente de D_G em tempo $O(1)$.*

Prova. Para localizar um componente $D_G(i,k)$ na estrutura, primeiro é calculado em que subvetor de D_G^i está o componente. É fácil notar que a posição inicial deste subvetor em DR_G é dada por $Loc(i, \lfloor k/s \rfloor)$. Acrescentando o deslocamento (k mod s) para encontrar o componente dentro do subvetor fica claro que $D_G(i,k) = DR_G(Loc(i, \lfloor k/s \rfloor) + k \bmod s)$. Todas as operações indicadas podem ser feitas em tempo constante. □

Descrevemos agora a construção de DR_G e Loc. A linha D_G^0 é a primeira a ser incluída em DR_G. Cada subvetor de comprimento s de D_G^0 é alocado em um trecho de DR_G de comprimento $2s$, sendo que o espaço extra é usado para as próximas linhas de D_G. Assim, o vetor Loc^0 é prenchido com valores múltiplos de $2s$ e cada subvetor de D_G^0 é seguido por um espaço vago de tamanho s.

A construção da estrutura é feita de forma incremental, incluindo-se os dados de uma linha de D_G por vez. O que possibilita que DR_G contenha todos os dados de D_G ocupando apenas

espaço $O(n_b s)$ é o fato de que os subvetores de diferentes linhas de D_G podem se sobrepor. Se os dados da linha D_G^i já foram incluídos, a inclusão dos dados de D_G^{i+1} é feita aproveitando-se boa parte dos dados já presentes na estrutura.

Como já mostrado na Propriedade 4.3 e na Definição 4.3, a única diferença entre as linhas D_G^i e D_G^{i+1} é que D_G^{i+1} não possui o valor $D_G^i(0) = i$ mas pode possuir um valor que não pertence a D_G^i, dado por $V_G(i+1)$. A construção da representação de D_G^{i+1} pode ser assim esboçada:

1. Determina-se em qual dos s subvetores de D_G^i o componente $V_G(i+1)$ deve ser inserido. Seja v o índice deste subvetor.
2. Determina-se em qual posição deste subvetor deve ser inserido $V_G(i+1)$.
3. Todos os subvetores de D_G^{i+1} de índice *superior* a v são iguais aos de D_G^i, não havendo necessidade de reescrevê-los em DR_G. Basta fazer $Loc(i+1, j) = Loc(i, j)$ para $v < j < s$.
4. Todos os subvetores de D_G^{i+1} de índice *inferior* a v são iguais aos de D_G^i, a menos de um deslocamento para a esquerda e pela inclusão de um novo componente à direita (justamente aquele que foi "jogado para fora" do subvetor seguinte). A maneira mais simples de deslocar os subvetores para a esquerda é fazer $Loc(i+1, j) = Loc(i, j) + 1$ para $0 \leq j < v$. O componente novo de cada subvetor pode ser escrito logo à direita do subvetor, desde que haja espaço vago para isto em DR_G. Os detalhes desta operação serão mostrados mais à frente.
5. O próprio subvetor de índice v sofre modificações maiores, de forma que um novo subvetor deve ser alocado em DR_G, já com a inclusão de $V_G(i+1)$. $Loc(i+1, v)$ indica a posição deste novo subvetor.

Este procedimento é ilustrado na Figura 4.5.

A cada vez que uma nova linha é incluída em DR_G um espaço vago de tamanho entre 0 e s é deixado após cada um dos s subvetores. Na inclusão da linha seguinte parte deste espaço pode ser ocupada, o que significa que poderá haver menos espaço vago após os subvetores da nova linha. Eventualmente, o procedimento descrito anteriormente não irá funcionar por falta de espaço para realização do passo 4. Quando isto acontece, um espaço inteiramente novo precisa ser alocado em DR_G.

Assim sendo, na inclusão de D_G^{i+1} há dois casos em que o algoritmo de construção requer a alocação de um novo espaço e a cópia de um subvetor inteiro:

1. O subvetor é aquele que deve conter $V_G(i+1)$. Ele deve ser copiado para uma nova região, já com $V_G(i+1)$ inserido.
2. Não é possível fazer o deslocamento para a esquerda de um subvetor de D_G^i por falta de espaço para a inclusão do componente extra no final. O subvetor correspondente de D_G^{i+1} é copiado, já deslocado e com o componente extra, em uma nova região.

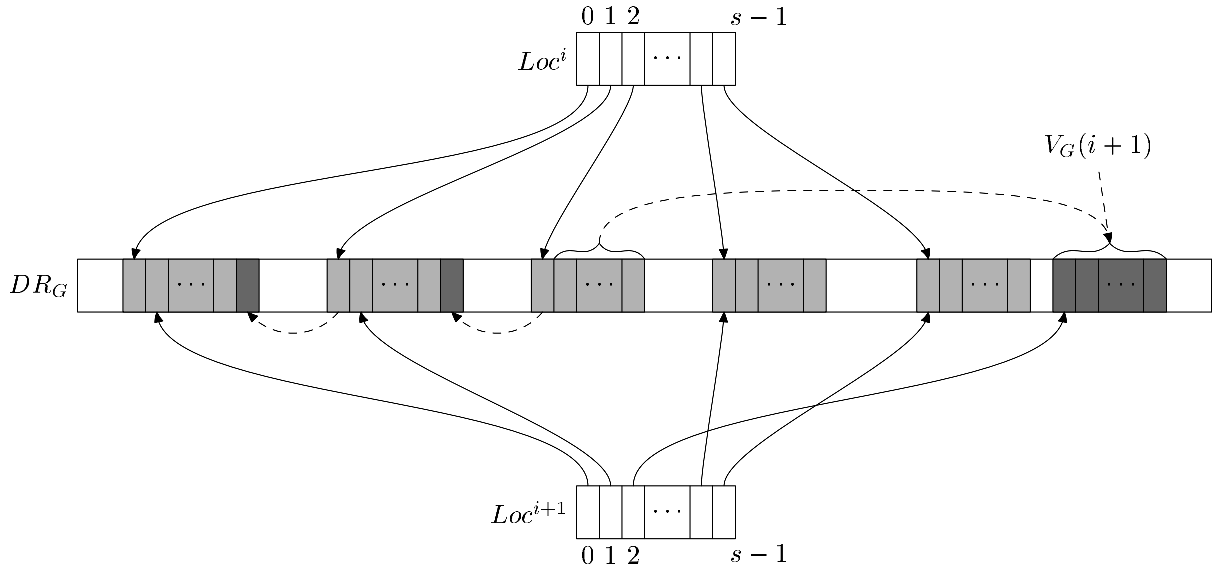

Figura 4.5: Inclusão de D_G^{i+1} em DR_G a partir dos dados referentes a D_G^i. Neste exemplo, o componente $V_G(i+1)$ deve ser inserido no subvetor de índice 2. As áreas mais escuras representam os dados escritos em DR_G nesta etapa. As setas tracejadas indicam cópias de dados.

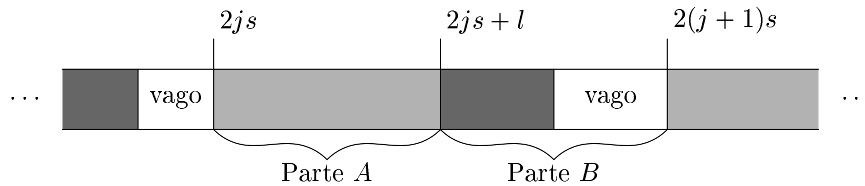

Figura 4.6: Unidade de alocação em DR_G. A parte A (tamanho s) é preenchida com dados copiados logo que a unidade é alocada. A parte B (também de tamanho s) é preenchida item por item quando novas linhas de D_G são incluídas. Quando cheia, a parte B é copiada na parte A de uma nova unidade de alocação.

Toda nova região alocada tem tamanho $2s$, sendo que o subvetor copiado nela ocupa apenas as s primeiras posições. O espaço vago adicional é aproveitado nas próximas inclusões de linhas. Cada nova linha pode requerer uma nova casa neste espaço vago, e quando a próxima casa a ser ocupada tiver índice múltiplo de $2s$ sabe-se que o espaço vago está esgotado. Este esquema de alocação é mostrado na Figura 4.6.

O Algoritmo 4.2 contém a descrição completa da construção da estrutura. Neste algoritmo, a variável *novo* indica o próximo trecho de comprimento $2s$ que está vago para alocação em DR_G.

Algoritmo 4.2: **Construção da Representação Compacta de D_G.**
Entrada: Vetores D_G^0 e V_G.
Saída: Vetor DR_G e matriz *Loc*.

1 $novo \leftarrow 0$

2 (* Inclui os dados de D_G^0 na estrutura *)
Para $v \leftarrow 0$ até $s-1$ faça

2.1 $Loc^0(v) \leftarrow novo$

2.2 (* Inclui os dados do subvetor de índice v *)
Para $k \leftarrow vs$ até $\min\{n_a, (v+1)s-1\}$ faça

2.2.1 $DR_G(Loc^0(v)+k-vs) \leftarrow D_G^0(k)$

2.3 $novo \leftarrow novo + 2s$

3 (* Inclui as demais linhas de D_G *)
Para $i \leftarrow 0$ até $n_b - 1$ faça

3.1 (* Encontra o subvetor que deve conter $V_G(i+1)$ *)
$v \leftarrow \min\limits_{1 \le j \le s}\{j | j = s \text{ ou } DR_G(Loc^i(j)) > V_G(i+1)\} - 1$

3.2 (* Encontra a posição de inserção de $V_G(i+1)$ no subvetor *)
$r \leftarrow \max\limits_{1 \le j < s}\{j | DR_G(Loc^i(v)+j) < V_G(i+1)\}$

3.3 (* aloca novo subvetor *)
$Loc^{i+1}(v) \leftarrow novo$

3.4 (* transfere os dados anteriores a r, deslocando-os para a esquerda *)
Para $j \leftarrow 0$ até $r-1$ faça

3.4.1 $DR_G(novo+j) \leftarrow DR_G(Loc^i(v)+j+1)$

3.5 (* Insere $V_G(i+1)$ *)
$DR_G(novo+r) \leftarrow V_G(i+1)$

3.6 (* transfere os dados posteriores a r *)
Para $j \leftarrow r+1$ até $s-1$ faça

3.6.1 $DR_G(novo+j) \leftarrow DR_G(Loc^i(v)+j)$

3.7 $novo \leftarrow novo + 2s$

3.8 (* compartilha subvetores de D_G^i e D_G^{i+1} acima de v *)
Para $j \leftarrow v+1$ até $s-1$ faça

3.8.1 $Loc^{i+1}(v) \leftarrow Loc^i(v)$

3.9 (* abaixo de v, compartilha subvetores se possível *)
Para $j \leftarrow 0$ até $v-1$ faça

3.9.1 (* verifica se o espaço vago após o subvetor acabou *)
Se $Loc^i(j) + s \equiv 0 \pmod{2s}$ então

3.9.1.1 (* aloca novo vetor e copia dados, deslocando para a esquerda *)
$Loc^{i+1}(j) \leftarrow novo$

3.9.1.2 Para $t \leftarrow 0$ até $s-2$

3.9.1.2.1 $DR_G(novo+t) \leftarrow DR_G(Loc^i(j)+t+1)$

3.9.1.3 $novo \leftarrow novo + 2s$

3.9.2 Senão

3.9.2.1 $Loc^{i+1}(j) \leftarrow Loc^{i}(j) + 1$

3.9.3 (* copia o primeiro dado do subvetor seguinte no final deste subvetor *)
$DR_G(Loc^{i+1}(j) + s - 1) \leftarrow DR_G(Loc^{i}(j))$

fim do algoritmo.

Lema 4.3 *A partir de D_G^0 e V_G, uma representação de D_G baseada em DR_G e Loc pode ser construída em tempo $O(n_b s)$ ocupando espaço $O(n_b s)$.*

Prova. Como há $n_b + 1$ linhas em D_G e cada uma requer espaço s em *Loc*, fica claro que *Loc* ocupa espaço $O(n_b s)$. Vamos nos preocupar apenas com o espaço ocupado em DR_G.

A informação contida em D_G^0 é processada no passo 2 em tempo $O(s^2)$ e ocupa espaço $O(s^2)$. A seguir, o laço do passo 3 inclui os dados de cada linha adicional de D_G em n_b iterações. Resta provar que em média cada iteração acrescenta apenas $O(s)$ dados a DR_G e executa em tempo $O(s)$. Vamos analisar cada comando deste laço.

Os passos 3.1 e 3.2 podem ser realizados com busca binária em tempo $O(\log s)$. Os passos 3.3 a 3.6 envolvem apenas laços que executam $O(s)$ iterações de tempo constante, portanto consomem tempo $O(s)$. Uma nova unidade de alocação de tamanho $2s$ é alocada e s dados são copiados. O passo 3.8 executa em tempo $O(s)$ e não acrescenta novos dados a DR_G.

A análise do passo 3.9 é um pouco diferente. Na verdade, os laços das linhas 3.9 e 3.9.1.2 juntos podem precisar de tempo e espaço maior do que $O(s)$. Isto pode ocorrer se vários subvetores ficarem sem espaço vago ao mesmo tempo.

Procedemos então com uma análise amortizada. Sempre que é preciso copiar um subvetor em outro no passo 3.9.1.2, a cópia é feita a partir da Parte B de uma unidade de alocação para a Parte A de uma unidade de alocação recém alocada (ver Figura 4.6). Se o custo da cópia de cada item for contabilizado "com antecedência" no momento em que o item for escrito pela primeira vez na Parte B (no passo 3.9.3), podemos considerar nulo o custo do passo 3.9.1.2. Por outro lado, o passo 3.9.3 deve ter custo dobrado, mas isto não altera a análise. Assim, o tempo (amortizado) do passo 3.9 é $O(s)$. A quantidade (amortizada) de dados acrescentados é também $O(s)$.

Assim, todos os passos no interior do laço 3 requerem tempo e espaço $O(s)$. O laço inteiro requer tempo e espaço $O(n_b s)$, o que conclui a análise. □

Algumas melhorias podem ser feitas no Algoritmo 4.2. Por exemplo, o caso em que $V_G(i+1) = \infty$ pode ser tratado de maneira diferenciada. No entanto, como estas melhorias não alteram os resultados da análise realizada, elas foram ignoradas por simplicidade.

Os resultados desta seção podem ser resumidos no seguinte teorema, cuja veracidade é conseqüência dos Lemas 4.2 e 4.3:

Teorema 4.4 *Sendo G o GDAG do problema ALCS para as cadeias A e B, a partir de D_G^0 e V_G é possível reconstruir uma representação de D_G em tempo e espaço $O\left(\sqrt{n_a} n_b\right)$ tal que qualquer valor de D_G pode ser lido em tempo $O(1)$.*

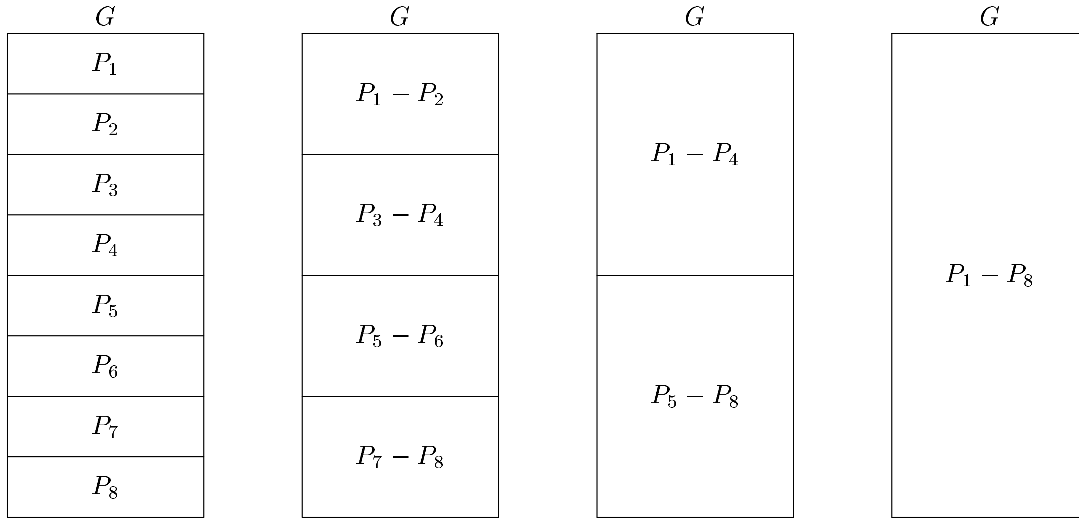

Figura 4.7: União de soluções parciais do ALCS, com $p = 8$. Em cada faixa do GDAG G estão indicados os processadores usados na solução do ALCS da faixa.

4.4 Idéia Básica do Algoritmo CGM para o Problema ALCS

Ao longo desta seção será mostrado um algoritmo paralelo CGM para o problema ALCS. Com p processadores, o algoritmo requer tempo $O(\frac{n_a n_b}{p})$ e $O(C \log p)$ rodadas de comunicação, sendo que em cada rodada cada processador envia/recebe $O(n_a p^{1/C} + n_b)$ dados. Nesta expressões, C é uma constante inteira (maior do que 0) a ser escolhida. A complexidade de tempo para este algoritmo é adequada para o seu uso na solução do problema LCS, mais simples. Vamos supor, por simplicidade e sem prejuízo para as análises de complexidade que se seguirão, que n_a é múltiplo de p e p é uma potência de 2.

O algoritmo envolve a divisão da cadeia A em p subcadeias de tamanho $\frac{n_a}{p}$ que não se sobrepõem. Para $1 \leq t \leq p$, o processador P_t resolve seqüencialmente o problema ALCS para as cadeias $A_{n_a(t-1)/p+1}^{n_a t/p}$ e B. O GDAG do problema original é dividido horizontalmente em faixas, obtendo-se p GDAGs de $\frac{n_a}{p} + 1$ linhas. Duas faixas contíguas compartilham uma linha.

O algoritmo seqüencial usado nesta etapa inicial é mostrado na Seção 4.2. O tempo necessário para os p processadores resolverem todos os subproblemas em paralelo é $O\left(\frac{n_a n_b}{p}\right)$, como mostrado no Teorema 4.1.

Em seguida são realizadas $\log p$ rodadas de união, em que pares de soluções parciais (para duas faixas vizinhas) são reunidas em uma única solução para a união das duas faixas. A cada etapa de união, o número de processadores associados a cada faixa dobra. Após estas $\log p$ rodadas temos a solução para o problema original. O tempo total de todas estas rodadas é $O\left(n_b\sqrt{n_a}\left(1 + \frac{\log n_a}{\sqrt{p}}\right)\right)$, como será mostrado na Seção 4.6. A Figura 4.7 ilustra o processo de união.

A parte mais complexa do algoritmo envolve o processo de união de duas faixas. Este processo será explicado na Seção 4.5.

4.5 União de Soluções Parciais

Nesta seção iremos estudar o processo de solução do ALCS para duas cadeias A_1^{2m} e B, de comprimento $2m$ e $n = n_b$ respectivamente, a partir das soluções do ALCS para A_1^m e B e para A_{m+1}^{2m} e B. O GDAG relativo ao primeiro subproblema será denominado S (superior) e o relativo ao segundo problema será I (inferior). O GDAG formado pela união destes dois GDAGs (pela fusão das linhas F_S e T_I) será denominado U.

Naturalmente, todas as propriedades já descritas para o GDAG G se aplicam a estes GDAGs parciais. A notação já introduzida para G será adaptada para estes GDAGs nas descrições seguintes.

Como explicado na Seção 4.1, os dados relacionados a um GDAG G podem ser armazenados em espaço reduzido usando os vetores D_G^0 e V_G. Assim, no início do processo de união as soluções para os subproblemas estarão disponíveis nos vetores D_S^0, D_I^0 (com índices entre 0 e m), V_S e V_I (com índices entre 1 e n).

Um total de q processadores estarão envolvidos na determinação de D_U^0 (índices de 0 a $2m$) e V_U (índices de 1 a n). Assim, os parâmetros para análise do processo de união são m, n e q. Por simplicidade, vamos supor que $m \leq n$. Se $m > n$, as linhas de D_S e D_I apresentariam ∞ em todas as posições superiores a n, o que poderia ser usado para limitar o tempo e espaço utilizados no processo de união.

Como mostrado na Seção 4.3, é possível acessar qualquer dado de D_S ou D_I em tempo constante, se for construída (em tempo e espaço $O(\sqrt{m}n)$) uma estrutura de dados própria para isto. Vamos então considerar que D_S e D_I estão disponíveis para acesso rápido.

4.5.1 Princípio Básico do Processo de União de Soluções

O princípio básico usado aqui é o mesmo que foi usado em [32]. Para cada i, $0 \leq i \leq n$, resolve-se o subproblema de determinar D_U^i a partir dos dados disponíveis.

Relembrando, $D_U^i(k) = D_U(i,k)$ representa o menor valor de j tal que $C_U(i,j)$, o peso total do melhor caminho entre $T_U(i)$ (vértice i do topo do GDAG U) e $F_U(j)$ (vértice j do fundo do GDAG U), é k. Todos os caminhos a partir de $T_U(i) = T_S(i)$ até $F_U(j) = F_I(j)$ têm que cruzar a fronteira comum $F_S = T_I$ em algum vértice e o peso total do caminho é a soma dos pesos dos trechos em S e em I. Assim, se estamos interessados em determinar $D_U(i,k)$ é preciso considerar caminhos que atravessam S com peso total l e depois atravessam I com peso total $k-l$, para todo l entre 0 e m inclusive.

Fixando um determinado valor de l, o menor valor de j tal que há um caminho de $T_U(i)$ a $F_U(j)$ com peso l em S e peso $k-l$ em I é dado por $D_I(D_S(i,l), k-l)$, pois $D_S(i,l)$ é o primeiro

vértice na fronteira que está a uma distância l de $T_U(i)$ e $D_I(D_S(i,l), k-l)$ é o primeiro vértice que está a uma distância $k-l$ do vértice na fronteira. As propriedades 4.2 justificam esta escolha.

Pelas considerações anteriores temos então que:

$$D_U(i,k) = \min_{0 \leq l \leq m} \{D_I(D_S(i,l), k-l)\}. \quad (4.1)$$

Deve-se observar que se mantivermos i fixo e variarmos k, as linhas de D_I utilizadas são sempre as mesmas. A variação de k faz mudar apenas o componente que deve ser consultado em cada linha.

Para cada linha de D_I consultada toma-se um componente diferente, devido ao termo $-l$. Este "deslocamento" sugere a seguinte definição:

Definição 4.7 ($Desl[l, W, c]$) *Dado um vetor W de comprimento $s+1$ (índices de 0 a s), para qualquer l ($0 \leq l \leq c-s$) define-se $Desl[l, W, c]$ como sendo o vetor de comprimento $c+1$ (índices de 0 a c) tal que:*

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

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

Com base nesta definição podemos reescrever a Equação 4.1:

$$D_U(i,k) = \min_{0 \leq l \leq m} \{Desl[D_I^{D_S(i,l)}, l, 2m](k)\} \quad (4.2)$$

Tomando todas as linhas relativas a um certo valor de i podemos, através das definições a seguir, montar uma matriz que servirá para encontrar *todos* os componentes de D_U^i.

Definição 4.8 ($Diag[W, M, l_0]$) *Seja W um vetor (índice inicial 0) de números inteiros em ordem crescente tal que os $m'+1$ primeiros componentes são finitos e $W(m') \leq n$. Seja M uma matriz $(n+1) \times (m+1)$ (índices iniciais 0). $Diag[W, M, l_0]$ é uma matriz $(m'+1) \times (2m+1)$ tal que a sua linha de índice l é $Desl[M^{W(l)}, l+l_0, 2m]$.*

$Diag[W, M, l_0]$ tem suas linhas copiadas a partir de uma matriz M. A seleção de quais linhas são copiadas é feita pelo vetor W. Cada linha copiada é deslocada uma coluna para a direita em relação à linha anterior. O deslocamento da primeira linha copiada é indicado por l_0. A Figura 4.8 ilustra esta construção.

Definição 4.9 ($MD[i]$) *Seja U um GDAG para o problema ALCS, formado pela união dos GDAGs S (superior) e I (inferior). Então $MD[U, i] = Diag[D_S^i, D_I, 0]$. Estando U claro no contexto, usaremos apenas a notação $MD[i]$.*

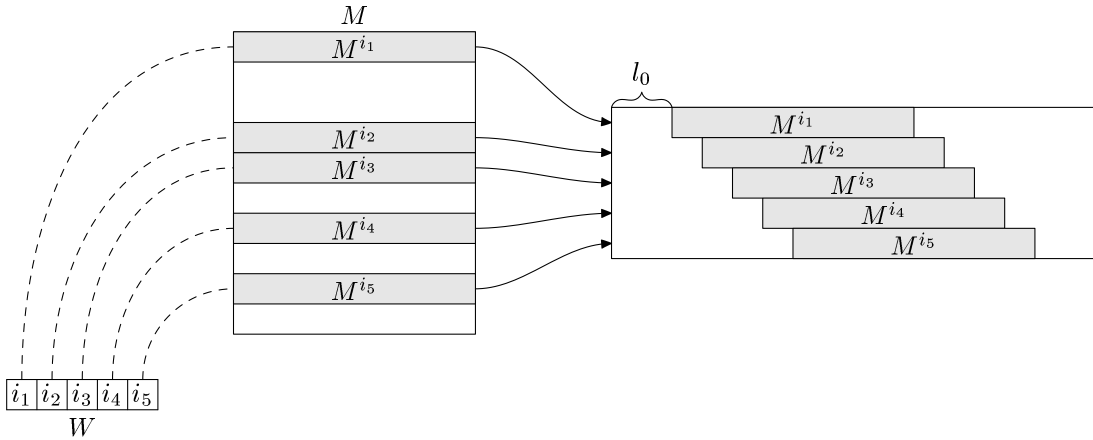

Figura 4.8: Construção de $Diag[W, M, l_0]$.

	k											
	0	1	2	3	4	5	6	7	8	9	10	11-16
$MD[2](0,k)$	2	3	4	5	6	8	9	13	∞	∞	∞	∞
$MD[2](1,k)$	∞	3	4	5	6	8	9	11	13	∞	∞	∞
$MD[2](2,k)$	∞	∞	4	5	6	8	9	11	13	∞	∞	∞
$MD[2](3,k)$	∞	∞	∞	5	6	7	8	9	11	13	∞	∞
$MD[2](4,k)$	∞	∞	∞	∞	6	7	8	9	11	13	∞	∞
$MD[2](5,k)$	∞	∞	∞	∞	∞	8	9	10	11	13	∞	∞
$MD[2](6,k)$	∞	∞	∞	∞	∞	∞	9	10	11	12	13	∞
$MD[2](7,k)$	∞	∞	∞	∞	∞	∞	∞	13	∞	∞	∞	∞

Tabela 4.5: $MD[2]$ considerando um GDAG formado pela união de dois GDAGs iguais ao da Figura 4.1.

Como exemplo, supondo que um GDAG U fosse formado por duas cópias do GDAG da Figura 4.1, D_S e D_I seriam iguais e dadas pela Tabela 4.2. $MD[2]$ teria suas linhas definidas por $D_I^2 = (2, 3, 4, 5, 6, 8, 9, 13, \infty)$ e sua estrutura completa seria a apresentada na Tabela 4.5. Note que não há linha definida a partir de $D_I(2, 8) = \infty$.

Pelas definições anteriores, resolvendo-se o Problema dos Mínimos das Colunas em $MD[i]$ encontra-se D_U^i. Mais exatamente, relembrando que $Cmin[M]$ é o vetor de mínimos das colunas da matriz M (Definição 2.1, página 21), temos:

$$D_U^i(k) = Cmin[MD[i]](k) \tag{4.3}$$

Continuando o exemplo da Tabela 4.5, o vetor D_U^2 seria $(2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, \infty)$.

Na Seção 2.3 o Problema dos Mínimos das Colunas foi abordado para a classe das matrizes totalmente monotônicas. O Teorema 4.5, enunciado mais à frente, indica que os resultados lá

apresentados podem ser aplicados aqui. No entanto, antes de enunciar este teorema é preciso definir como fazer comparações entre componentes infinitos de $MD[i]$ e estudar como estes componentes estão dispostos na matriz.

Observação 4.1 *Dados dois componentes infinitos de uma coluna j de $MD[i]$, $MD[i](l_1, j)$ e $MD[i](l_2, j)$, sendo $l_1 < l_2$, considera-se que $MD[i](l_1, j) > MD[i](l_2, j)$ se e somente se $l_2 < j$ (ou seja, os dois componentes estão sobre a diagonal principal). Caso contrário, considera-se que $MD[i](l_1, j) < MD[i](l_2, j)$.*

Observação 4.2 *Como todas as linhas de D_I têm o primeiro componente finito (mais precisamente $D_I(k, 0) = k$), toda a diagonal principal de $MD[i]$ é finita. Por construção, todos os componentes abaixo desta diagonal principal são infinitos. Além disso, os componentes finitos de qualquer linha de $MD[i]$ ocupam posições adjacentes, podendo haver componentes infinitos à esquerda e à direita.*

Teorema 4.5 *Fazendo comparações de acordo com a Observação 4.1, a matriz $MD[i]$ é totalmente monotônica.*

Prova. É preciso provar que toda submatriz 2×2 de $MD[i]$ é monotônica, ou seja, que para quaisquer $l_1 < l_2$ e $j_1 < j_2$, se $MD[i](l_2, j_1) < MD[i](l_1, j_1)$ então $MD[i](l_2, j_2) < MD[i](l_1, j_2)$.

Vamos estudar o caso em que $j_2 = j_1 + 1$ (usando duas colunas adjacentes em $MD[i]$). O caso mais geral pode ser facilmente provado indutivamente a partir deste caso.

Vamos supor inicialmente que todos os componentes envolvidos na submatriz são finitos. Sejam $v_1 = D_S(i, l_1)$ e $v_2 = D_S(i, l_2)$. Pela Propriedade 4.2(1) temos que $v_1 < v_2$. Usando as definições 4.7 e 4.9, a implicação que precisamos provar torna-se então

$$D_I(v_2, j_1 - l_2) < D_I(v_1, j_1 - l_1) \Rightarrow D_I(v_2, j_1 - l_2 + 1) < D_I(v_1, j_1 - l_1 + 1) \ .$$

Supondo que $D_I(v_2, j_1 - l_2) < D_I(v_1, j_1 - l_1)$ e lembrando que todos os componentes de $D_I^{v_2}$ são maiores que ou iguais a v_2 (decorre de imediato da definição) temos que $D_I(v_1, j_1 - l_1) > v_2$. Assim sendo, a Propriedade 4.2(3) garante que $t = D_I(v_1, j_1 - l_1)$ é componente comum a $D_I^{v_1}$ e $D_I^{v_2}$.

Deve-se observar que a Propriedade 4.2(1) indica que $D_I^{v_2}$ é crescente, logo não deve haver em $D_I^{v_2}$ nenhum componente com valor entre os componentes adjacentes $D_I(v_2, j_1 - l_2)$ e $D_I(v_2, j_1 - l_2 + 1)$. Como $D_I(v_2, j_1 - l_2) < t$ é preciso ter $D_I(v_2, j_1 - l_2 + 1) < t$. Como $t < D_I(v_1, j_1 - l_1 + 1)$ (novamente pela Propriedade 4.2(1)), concluímos que $D_I(v_2, j_1 - l_2 + 1) < D_I(v_1, j_1 - l_1 + 1)$ como esperado. A Figura 4.9 ilustra esta demonstração.

Para estudar os casos em que há pelo menos um componente infinito na submatriz, utilizamos as observações 4.1 e 4.2. Considerando os 4 componentes da submatriz, há ao todo 16 possíveis colocações de componentes infinitos. Eliminamos o caso (já estudado) em que não há componentes infinitos. Eliminamos também os casos em que a submatriz é claramente monotônica. Restam então os seguintes casos:

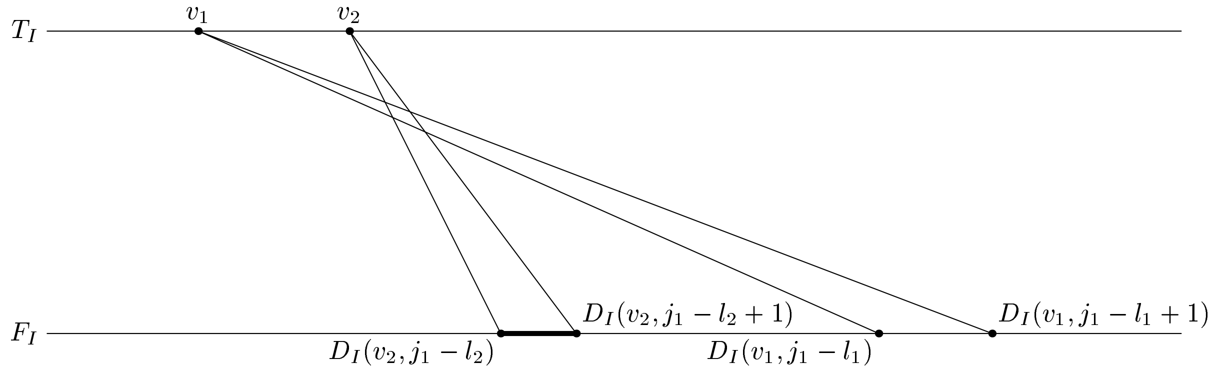

Figura 4.9: Demonstração de monotonicidade de submatriz 2×2 de $MD[i]$ quando todos os componentes são finitos. A figura ilustra o GDAG I apenas. O trecho marcado em F_I está compreendido entre dois componentes adjacentes de $D_I^{v_2}$, o que impede que nele exista um ponto de quebra para qualquer vértice anterior a v_2.

$$\underset{\text{(A)}}{\begin{bmatrix} \infty & a \\ b & c \end{bmatrix}} \quad \underset{\text{(B)}}{\begin{bmatrix} \infty & a \\ b & \infty \end{bmatrix}} \quad \underset{\text{(C)}}{\begin{bmatrix} \infty & \infty \\ b & \infty \end{bmatrix}} \quad \underset{\text{(D)}}{\begin{bmatrix} \infty & a \\ \infty & \infty \end{bmatrix}} \quad \underset{\text{(E)}}{\begin{bmatrix} \infty & a \\ \infty & c \end{bmatrix}} \quad \underset{\text{(F)}}{\begin{bmatrix} \infty & \infty \\ \infty & \infty \end{bmatrix}}$$

Pela Observação 4.2 os casos (A) e (B) não podem acontecer. O caso (C) só pode acontecer se a segunda coluna da submatriz contiver componentes acima da diagonal principal de $MD[i]$, logo o mínimo da segunda coluna está na segunda linha (Observação 4.1) e a submatriz é monotônica.

Os casos seguintes também envolvem a Observação 4.1. Nos casos (D) e (E) a primeira coluna contém componentes que estavam abaixo da diagonal principal de $MD[i]$, logo o mínimo desta coluna está na primeira linha e a submatriz é monotônica (na verdade, o caso (E) não é possível quando consideramos apenas submatrizes tomadas de colunas adjacentes de $MD[i]$). No caso (F) a submatriz é monotônica porque se a primeira coluna contiver componentes que estão acima da diagonal principal de $MD[i]$ o mesmo acontecerá com a segunda coluna.

Assim, todas as possíveis submatrizes 2×2 de $MD[i]$ foram consideradas: as tomadas de colunas adjacentes foram verificadas diretamente e pode-se provar que as demais são monotônicas por indução finita. $\square$

Dado que todas as matrizes $MD[i]$ são totalmente monotônicas, o problema da união de GDAGs pode ser resolvido através da busca pelo mínimo das colunas em todas elas. O Algoritmo 2.5 permite que estas buscas sejam feitas em tempo $O(m)$ para cada uma das $n + 1$ matrizes (pois as matrizes têm altura m), totalizando tempo $O(nm)$. Isto não é bom o bastante.

Para resolver o problema da união em tempo menor é preciso observar que, dadas as semelhanças entre vetores adjacentes de D_S, matrizes $MD[i]$ para valores próximos de i são também muito semelhantes. Isto será explorado a seguir.

4.5.2 Eliminação de Redundâncias entre Subproblemas

Ainda explorando as técnicas apresentadas em [32], usamos o fato de que um conjunto de $r+1$ linhas adjacentes de D_G são r-variantes: é possível obter uma linha a partir de outra através de no máximo r inserções e r remoções de componentes. Mais importante: com r linhas de comprimento m é possível determinar um vetor de componentes comuns de tamanho $m-r$, que chamaremos simplesmente de *vetor comum*. Usaremos a notação $D_G^{i_0,r}$ para indicar o vetor comum às linhas entre $D_G^{i_0}$ e $D_G^{i_0+r}$ (inclusive) para um GDAG G qualquer.

Pela Propriedade 4.2(3), todos os componentes de um vetor $D_G^{i_0}$ estarão presentes no vetor $D_G^{i_0+r}$, menos os que são menores do que i_0+r. Por exemplo, usando dados da Tabela 4.2, temos que os componentes comuns a $D_G^0 = (0,1,2,3,4,5,6,8,9)$ e $D_G^6 = (6,7,8,9,11,13,\infty,\infty,\infty)$ são 6, 8 e 9 (os últimos de D_G^0). Usando matrizes maiores e valores de r significativamente inferiores a m, o número de componentes comuns torna-se próximo de m. Em [32] usa-se $r = \log^2 m$ e $r = \log^4 m$ em dois algoritmos distintos. O valor usado neste trabalho é $r = \lceil\sqrt{m}\rceil$.

Pela Propriedade 4.2(4), todos os componentes presentes em ambos os vetores $D_G^{i_0}$ e $D_G^{i_0+r}$ estarão presentes também nos vetores D_G^i para $i_0 < i < i_0+r$. Temos então uma forma simples de determinar o vetor comum para o grupo de linhas: basta consultar a matriz D_G e ler todos os componentes de $D_G^{i_0}$, descartando os que forem menores do que i_0+r. Esta cópia leva tempo $O(m)$. Desta forma, no exemplo anterior teremos $D_G^{i_0,r} = (6,8,9)$.

Além disso, é importante determinar quais componentes são adjacentes em todos os vetores, formando "trechos indivisíveis" que chamaremos *trechos comuns*. No exemplo anterior, os trechos comuns de $D_G^{i_0,r}$ são $(6,8)$ e (9).

Esta determinação pode ser feita com base nos dados do vetor V_G. De $V_G(i_0+1)$ até $V_G(i_0+r)$ temos todos os componentes que não aparecem no vetor comum e podem dividí-lo (Note que os componentes removidos de $D_G^{i_0}$ também não aparecem no vetor comum mas estão todos "à esquerda" dele). Determinamos as divisões entre os trechos procurando o ponto de inserção de cada um dos componentes de V_G, em tempo $O(\log m)$ por componente ou $O(r \log m)$ no total.

Com r possíveis pontos de divisão, fica claro que haverá no máximo $r+1$ trechos em $D_G^{i_0,r}$. Estes trechos serão numerados de 0 a r e o trecho t será denominado $D_G^{i_0,r}[t]$.

Estas operações de extração de vetor comum e determinação de trechos comuns será aplicada à matriz D_S em cada etapa de união do algoritmo. As linhas de $D_S^{i_0}$ a $D_S^{i_0+r}$ norteiam a construção das matrizes de $MD[i_0]$ a $MD[i_0+r]$ e, como já foi mencionado, as semelhanças entre linhas próximas em D_S levam a semelhanças entre matrizes $MD[i]$ para valores próximos de i. O vetor comum $D_S^{i_0,r}$ contém índices das linhas de D_I que estarão presentes em todas as matrizes de $MD[i_0]$ a $MD[i_0+r]$. Cada trecho $D_S^{i_0,r}[t]$ indica um conjunto de linhas de D_I que serão usadas em linhas *adjacentes* em todas estas matrizes.

Consideremos um trecho comum $D_S^{i_0,r}[t]$, $0 \le t \le r$. Todas as matrizes de $MD[i]$ com $i_0 \le i \le i_0+r$ conterão $Diag[D_S^{i_0,r}[t], D_I, l_{i,t}]$ como um conjunto de linhas contíguas a partir da linha $l_{i,t}$, sendo que $l_{i,t}$ varia de matriz para matriz e de bloco para bloco. Isto é ilustrado na Figura 4.10.

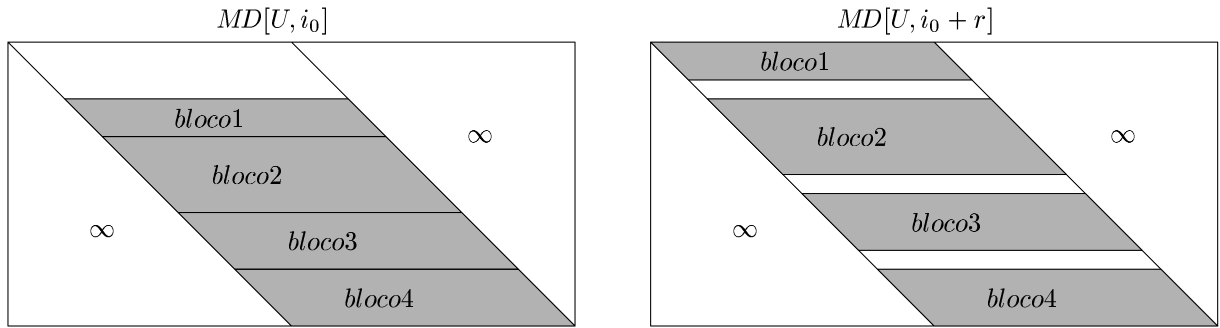

Figura 4.10: Estrutura das matrizes $MD[i_0]$ e $MD[i_0+r]$, destacando r blocos comuns (no caso, $r = 3$). Estes blocos estão presentes também nas matrizes entre $MD[i_0]$ e $MD[i_0 + r]$.

Estes blocos comuns cobrem a maior parte das matrizes $MD[i]$ quando $r \ll m$. Como em cada matriz será necessário resolver o Problema dos Mínimos das Colunas, determinar previamente os mínimos das colunas dos blocos comuns deve evitar o processamento repetido destes blocos. Temos então a seguinte definição:

Definição 4.10 ($ContBl[i_0, r, t]$)

$$ContBl[i_0, r, t] = Cmin[Diag[D_S^{i_0,r}[t], D_I, 0]]$$

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

A determinação dos mínimos das colunas dos blocos será coberta na Seção 4.5.3.

Lembramos então a definição da operação de *contrações de linhas* (Definição 2.5, página 30). Se para cada matriz $MD[i]$ realizarmos sucessivas *contrações de linhas*, uma para cada um dos blocos comuns, o resultado será uma matriz que chamaremos $ContMD[i]$, também totalmente monotônica e tal que $Cmin[MD[i]] = Cmin[ContMD[i]]$. Este resultado se deve ao Teorema 2.10.

Os blocos comuns aparecem nas diversas matrizes $MD[i]$ em posições diferentes, mas o resultado da busca pelos mínimos das colunas destes blocos pode ser aproveitada em todas as matrizes. Mais exatamente, se o bloco comum t aparece a partir da linha $l_{i,t}$ da matriz $MD[i]$, a contração deste bloco nesta matriz é feita pela simples substituição do bloco pelo vetor $Desl[l_{i,t}, ContBl[i_0, r, t], 2m]$. A forma pela qual esta substituição é realizada é mostrada na Seção 4.5.4.

Cada matriz $ContMD[i]$ conterá, além das $r + 1$ linhas provenientes da contração dos blocos comuns, no máximo r linhas adicionais. Como já comentado, $r = \lceil \sqrt{m} \rceil$ e portanto a contração das matrizes reduz a altura destas a $O(\sqrt{m})$. O benefício de se operar sobre $r + 1 = \lceil \sqrt{m} \rceil + 1$ matrizes de altura reduzida compensa o custo adicional de operar inicialmente sobre os blocos comuns.

As seções seguintes irão detalhar e analisar cada passo do procedimento.

4.5.3 Determinação dos Mínimos das Colunas dos Blocos Comuns

Nesta seção abordamos o problema da determinação dos vetores $ContBl[i_0, r, t]$, $0 \leq t \leq r$. Cada um destes vetores contém a solução para o Problema dos Mínimos das Colunas de uma matriz $Diag[D_S^{i_0,r}[t], D_I, 0]$. Cada matriz tem largura $\Theta(m)$ e a soma das alturas de todas elas é $O(m)$ (o comprimento do vetor comum $D_S^{i_0,r}$).

Como já foi comentado, os $r+1$ trechos comuns de $D_S^{i_0,r}$ podem ser determinados em tempo $O(m)$. Chamaremos m_t o comprimento do trecho comum t, que define a altura da matriz $Diag[D_S^{i_0,r}[t], D_I, 0]$.

Cada componente de cada trecho determinado pode ser acessado em tempo $O(1)$, assim como cada componente das matrizes D^I e D^S, logo podemos considerar que os componentes das matrizes podem ser acessados (indiretamente) em tempo $O(1)$.

O Algoritmo 2.5 para a resolução do Problema dos Mínimos das Colunas para matrizes totalmente monotônicas foi mostrado na Seção 2.3.2. Este algoritmo encontra o vetor de soluções para o bloco t em tempo $O(m_t + m + m_t \log(m/m_t)))$ (Teorema 2.8). O termo m se deve à necessidade de tornar explícito o vetor de respostas. Com isto, o tempo para determinação de todos os r vetores $ContBl[i_0, r, t]$ é $\Theta(mr) = \Theta(m\sqrt{m})$, excessivo para a obtenção de uma aceleração linear para o problema ALCS.

O Algoritmo 2.5 pode ser adaptado como sugerido na Seção 2.3.2 (Corolário 2.9) para montar uma estrutura de dados que permita a consulta aos mínimos das colunas de $Diag[D_S^{i_0,r}[t], D_I, 0]$ em tempo $O(\log m_t) = O(\log m)$. O tempo maior para consulta a este mínimos é compensado pelo menor tempo de construção da estrutura, $O(m_t \log(m/m_t))$. O tempo para construção das estruturas de todas as matrizes é $O(m \log m)$.

No final desta construção, os vetores $ContBl[i_0, r, t]$ estarão disponíveis para consulta em tempo $O(\log m)$ por componente. O espaço utilizado para construção e manutenção de todos estes vetores é $O(m \log m)$ (Corolário 2.9).

4.5.4 Representação das Matrizes $ContMD[i]$

Uma vez determinados os mínimos das colunas dos blocos comuns, procede-se com a determinação dos mínimos das colunas de $ContMD[i]$. Estes mínimos formam a matriz D_U que queremos determinar.

Para descrever a solução destes problemas de minimização é preciso descrever inicialmente como $ContMD[i]$ ($i_0 \leq i \leq i_0 + r$) será representada e acessada. Todas as linhas destas matrizes já estão disponíveis, algumas de forma direta (através das linhas de D_I) outras de forma indireta (através da representação dos vetores $ContBl[i_0, r, t]$). Falta especificar como determinar quais linhas são usadas em quais matrizes. Também é preciso especificar o *deslocamento para a direita* que cada linha terá na matriz.

Como as matrizes $ContMD[i]$ serão processadas seqüencialmente, pode-se montar uma estrutura capaz de representar apenas uma matriz por vez, modificando-a a cada etapa para

representar a matriz seguinte. Dois vetores serão usados para isto, ambos de tamanho $2r + 1$ (índices de 0 a $2r$). O vetor Lin indicará a "fonte" (linha de D_I ou vetor $ContBl[i_0, r, t]$) dos dados de uma certa linha de $ContMD[i]$. O vetor Des indicará o quanto esta "fonte" é deslocada para a direita.

A montagem de $ContMD[i_0]$, a primeira matriz a ser usada no grupo, é feita diretamente a partir de $D_S^{i_0}$. Esta linha de D_S é lida item por item e os vetores Lin e Des são preenchidos a partir do índice 0. Cada componente de $D_S^{i_0}$ que *não faz* parte do vetor comum é inserido na posição seguinte de Lin, representando uma única linha de D_I. Quando um componente que *faz* parte do vetor comum é encontrado em $D_S^{i_0}$, ele e todos os demais componentes do mesmo *trecho comum* são substituídos em Lin por uma única referência ao vetor $ContBl[i_0, r, t]$ correspondente, representando um bloco de linhas já contraído.

A primeira posição de Des recebe o valor 0. Para $l > 0$, $Des(l)$ recebe $Des(l-1)+1$, caso o item registrado em $Lin(l-1)$ seja uma linha simples, ou $Des(l-1)+m_t$, caso seja um bloco de altura m_t contraído.

Esta construção pode ser feita em tempo $O(m)$. Para consulta a um componente qualquer $ContMD[i_0](l, k)$ procede-se da seguinte maneira: $Lin(l)$ indica a fonte dos dados, um vetor do qual se obtém o componente de índice $k - Des(l)$. Caso este índice não possa ser utilizado, o componente devolvido é ∞. O tempo total de acesso é $O(1)$ para linhas simples, $O(\log m)$ para blocos contraídos.

Uma vez utilizada a matriz $ContMD[i_0]$ e determinada a linha $D_U^{i_0} = Cmin[ContMD[i_0]]$ pode-se prosseguir para as demais matrizes. A modificação em Lin e Des para cada matriz $ContMD[i]$, $i_0 < i \leq i_0 + r$, é feita pela inserção de $V_S(i)$ em Lin. Todos os componentes menores do que $V_S(i)$ (incluindo os que representam blocos de linhas com índices menores do que $V_S(i)$) são deslocados uma posição para a esquerda. O que estava na posição 0 é descartado. As posições correspondentes em Des são decrementadas e a que corresponde a $V_S(i)$ é calculada. Tudo isso pode ser feito em tempo $O(r) = O(\sqrt{m})$.

O espaço adicional para representação das matrizes é $O(\sqrt{m})$ e o tempo total para manutenção desta representação ao longo do processamento de $r + 1$ matrizes é $O(m)$. Passamos a seguir para o estudo da utilização propriamente dita destas matrizes. De agora em diante, consideraremos o tempo de acesso a $ContMD[i](l, k)$ como sendo $O(\log m)$, não considerando mais os detalhes de implementação.

4.5.5 Determinação das linhas i_0 a $i_0 + r$ de D_U

A determinação da linha $D_U^{i_0}$ é feita a partir de $ContMD[i_0]$, usando o Algoritmo 2.5. Esta determinação leva tempo $O(\sqrt{m}\log m + m)$, sendo que o termo m se deve à necessidade de explicitar todos os componentes de $D_U^{i_0}$, enquanto que o fator $\log m$ se deve ao tempo de acesso a cada item de $ContMD[i_0]$. Ao longo desta seção, este fator $\log m$ irá aparecer em todas as estimativas de tempo.

Cada uma das linhas seguintes é gerada com base na linha anterior. Devido às Propriedades 4.3, tendo-se D_U^i obtém-se D_U^{i+1} pela remoção do primeiro componente e inserção de um

novo componente, $V_U(i+1)$, em ordem com os demais. Todos os componentes de D_U^i anteriores à posição de $V_U(i+1)$ são deslocados uma casa para a esquerda (o primeiro, como já foi dito, é removido). Os demais componentes se mantém inalterados.

A determinação de D_U^{i+1} pode se resumir à determinação de $V_U(i+1)$, o que evita a pesquisa pelo mínimo de *todas* as colunas de $ContMD[i+1]$. Se o mínimo de uma certa coluna k for igual a $D_U^i(k)$, isto significa que $V_U(i+1)$ deve ser inserido em uma posição de índice menor do que k. Se o mínimo da coluna k for maior do que $D_U(i,k)$, isto significa que $V_U(i+1)$ deve ser inserido na posição k ou superior.

Ao todo há $2m+1$ colunas em $ContMD[i+1]$. Tomamos então as colunas de índice múltiplo de $r = \lceil\sqrt{m}\rceil$ (exceto a de índice 0) e determinamos os mínimos de cada uma delas. Isto pode ser feito através do Algoritmo 2.5 para a submatriz $ContMD[i+1](l,k)[k = rs \leq 2m, s = 1, 2, \ldots, \lfloor 2m/r \rfloor]$. Esta submatriz tem menos de $2\sqrt{m}$ colunas e cerca de $\sqrt{m}$ linhas, logo o Algoritmo 2.5 executa em tempo $O(\sqrt{m}\log m)$.

Comparamos então os mínimos das colunas selecionadas com os componentes de D_U^i de índice múltiplo de r. Sendo s_0 o menor valor de s tal que $D_U(i,rs)$ é igual ao mínimo da coluna rs de $ContMD[i+1]$, sabemos que $V_U(i+1)$ deve estar em uma coluna entre $k_1 = r(s_0-1)$ e $k_2 = rs_0$, inclusive. Caso não exista tal s_0, $V_U(i+1)$ estará em uma coluna entre $k_1 = r\lfloor 2m/r \rfloor$ e $k_2 = 2m$, inclusive. A determinação de s_0 pode ser feita em tempo $O(\log m)$ via busca binária.

Usando novamente o Algoritmo 2.5, agora para a submatriz $ContMD[i+1](l,k)[k_1 \leq k \leq k_2]$, e comparando os mínimos das colunas com o trecho correspondente em D_U^i, obtemos a posição e o valor de $V_U(i+1)$. Esta etapa também consome tempo $O(\sqrt{m}\log m)$.

Concluímos então que a determinação das linhas i_0+1 a i_0+r de D_U pode ser feita em tempo $O(r\sqrt{m}\log m) = O(m\log m)$. Incluindo a determinação da linha i_0, ainda temos tempo $O(m\log m)$.

Uma observação se faz necessária aqui: este procedimento exige que a linha D_U^i esteja disponível para consulta em tempo $O(1)$ durante a determinação de $V_U(i+1)$. Isto pode ser feito usando a estrutura mostrada na Seção 4.3 para representar a matriz D_U. Inicialmente esta estrutura armazena $D_U^{i_0}$. Depois, a cada novo componente de V_U determinado, a estrutura é atualizada em tempo $O(\sqrt{m})$ para representar a nova linha determinada de D_U.

Como o interesse do processo de união de GDAGs é gerar apenas D_U^0 e V_U, as linhas de D_U adicionais podem ser descartadas à medida em que se tornam desnecessárias. A estrutura da Seção 4.3 pode ser facilmente modificada para ocupar apenas espaço $O(m)$ e conter apenas os componentes da última linha determinada de D_U.

4.5.6 Análise Completa do Processo de União

Aproveitando as redundâncias entre as matrizes $MD[i]$ é possível determinar $D_U^{i_0}$ e $V_U(i)$ para $i_0 \leq i \leq i_0+r$ em tempo $O(m\log m)$, sendo que $O(m\log m)$ é gasto na contração dos blocos de linhas comuns das matrizes $MD[i]$, $O(m)$ é gasto na manutenção das matrizes $ContMD[i]$ e $O(m\log m)$ é gasto na determinação dos resultados. Considerando que há ao todo $(n+1)/(r+1)$

grupos de $r+1$ matrizes $MD[i]$ para processar deste modo, o tempo total para determinação de D_U^0 e V_U a partir de D_S e D_I é $O((nm/r)\log m) = O(n\sqrt{m}\log m)$.

Pode-se dividir este trabalho por q processadores através da divisão de D_S entre os processadores. Cada bloco de r linhas de D_S pode ser usado para determinar r linhas de D_U, sendo que o processamento de cada bloco é independente dos demais blocos. Com isto, o tempo cai para $O((n\sqrt{m}\log m)/q)$.

Ainda é preciso considerar o tempo de construção da representação de D_S e D_I, conforme mostrado na Seção 4.3. O maior problema é a representação de D_I, que deve estar disponível na memória local de todos os processadores. Esta construção toma tempo $O(n\sqrt{m})$ (ou tempo $O(n\sqrt{m}/q)$ se for realizada em paralelo, sob a pena de aumentar o custo das rodadas de comunicação).

Com estes resultados, podemos citar o seguinte lema:

Lema 4.6 *Seja U um GDAG $(2m+1)\times(n+1)$ do problema ALCS, formado pela união dos GDAGs $(m+1)\times(n+1)$ S (superior) e I (inferior). A determinação de D_U^0 e V_U a partir de D_S^0, V_S, D_I^0 e V_I pode ser feita por q processadores em tempo $O\left(n\sqrt{m}\left(1+\frac{\log m}{q}\right)\right)$ e espaço $O(n\sqrt{m})$.*

4.6 Análise do Algoritmo CGM para o Problema ALCS

Os parâmetros do problema completo são os comprimentos das cadeias A e B e o número de processadores, respectivamente n_a, n_b e p. Como já citado, a resolução do ALCS nos p GDAGs definidos para p subcadeias de A leva tempo $O\left(\frac{n_a n_b}{p}\right)$.

As $\log p$ etapas de união de GDAGs que se seguem tomam tempo $O\left(n\sqrt{m}\left(1+\frac{\log m}{q}\right)\right)$ cada uma, como visto no Lema 4.6. Deve-se notar que $n = n_b$ e $m = \frac{n_a q}{2p}$, fazendo com que a expressão do tempo se torne

$$O\left(\frac{n_b\sqrt{n_a q}}{\sqrt{2p}}\left(1+\frac{\log\frac{n_a q}{2p}}{q}\right)\right) = O\left(\frac{n_b\sqrt{n_a}}{\sqrt{p}}\left(\sqrt{q}+\frac{\log n_a}{\sqrt{q}}\right)\right) .$$

Assim, o tempo de cada etapa de união é definido com uma única variável, que é o número de processadores envolvido em cada GDAG, q. Este número dobra a cada etapa de união, logo a soma do tempo de todas as etapas é

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

$$O\left(n_b\sqrt{n_a}\left(1+\frac{\log n_a}{\sqrt{p}}\right)\right) .$$

Para que o algoritmo CGM para o problema ALCS completo tenha aceleração linear é preciso que o tempo acima seja $O\left(\frac{n_a n_b}{p}\right)$. Pode-se verificar facilmente que isto é conseguido se $p < \sqrt{n_a}$.

O espaço necessário para execução do algoritmo CGM completo é $O(n_b\sqrt{n_a})$ por processador, devido à representação de D_I na última etapa de união.

Estudamos agora a complexidade das comunicações. Quando há q processadores trabalhando em uma união, cada um deles determina n_b/q componentes de V_U que precisa transmitir para outros $2q-1$ processadores que trabalharão na próxima etapa de união, o que resulta em $O(n_b)$ dados transferidos por processador.

O processador que determina D_U^0 precisa transferir os $\frac{n_a q}{p}$ componentes deste vetor para os outros $2q-1$ processadores, o que resulta em uma rodada de comunicação em que $O(\frac{n_a q^2}{p})$ dados são transferidos.

Para alguma constante C, isto também pode ser feito em C rodadas de comunicação em que cada processador transfere $O\left(\frac{n_a q^{1+1/C}}{p}\right)$ dados: na primeira rodada o processador que determinou D_U^0 faz o *broadcast* deste vetor para $\lfloor q^{1/C} \rfloor$ outros processadores, que na etapa seguinte transmitem para $\lfloor q^{2/C} \rfloor$ outros processadores e assim por diante. Há outros esquemas para redução do número de dados que um processador precisa transmitir, utilizando um maior número de rodadas. Vamos nos ater a este, que mantém o número de rodadas constante para cada etapa de união. A etapa de união de GDAGs com maior número de dados transmitidos por processador é a última, em que cada processador precisa transmitir $O(n_a p^{1/C} + n_b)$ dados, já considerando D_U^0 e V_U.

Na última etapa de união, os vetores D_G^0 e V_G são determinados para o GDAG G completo, que representa o problema ALCS. Com uma ligeira alteração, cada processador pode gerar a estrutura compacta que armazena parte da matriz D_G. Na verdade, esta estrutura já é gerada na Seção 4.5.5, mas não é mantida por economia de espaço. Alterando a última etapa para que a estrutura seja mantida, o processador P_t $(1 \leq t \leq p)$ terá a representação compacta de $D_G^{\lfloor n_b(t-1)/p \rfloor}$ a $D_G^{\lfloor n_b t/p \rfloor - 1}$.

Podemos finalmente enunciar o seguinte teorema, cuja demonstração já foi feita nesta seção:

Teorema 4.7 *Dadas duas cadeias A e B, respectivamente de comprimentos n_a e n_b, e sendo G o GDAG característico do Problema ALCS para A e B, a construção de D_G pode ser realizada por $p < \sqrt{n_a}$ processadores em tempo $O\left(\frac{n_a n_b}{p}\right)$, espaço $O(n_b\sqrt{n_a})$ por processador e $O(C \log p)$ rodadas de comunicação em que $O(n_a p^{1/C} + n_b)$ dados são transferidos de/para cada processador.*

A matriz D_G é suficiente para a determinação do valor de alinhamento entre a cadeia A e qualquer subcadeia de B, mas não em tempo $O(1)$. Para isto seria necessária a matriz C_G (Definição 4.1). Um acesso a $C_G(i,j)$ pode ser simulado em tempo $O(\log n_a)$, usando busca binária em D_G^i para encontrar o menor valor de k tal que $D_G^i(k)$ é maior do que j. O valor de $C_G(i,j)$ é $k-1$.

Caso um acesso mais rápido seja desejado, é preciso construir uma representação de C_G. O

processo de construção não será mostrado por ser muito simples: cada linha de D_G leva a uma linha de C_G em tempo $O(n_b)$. A construção completa, realizada por p processadores, leva tempo $O(n_b^2/p)$. O espaço requerido é também $O(n_b^2/p)$ por processador, o que pode representar um aumento com relação ao espaço usado ao longo do algoritmo.

Estes resultados são sumarizados no teorema a seguir:

Teorema 4.8 *Dadas duas cadeias A e B, respectivamente de comprimentos n_a e n_b, e sendo G o GDAG característico do Problema ALCS para A e B, a construção de C_G pode ser realizada por $p < \sqrt{n_a}$ processadores em tempo $O\left(\frac{(n_a+n_b)n_b}{p}\right)$, espaço $O(n_b \max\{\sqrt{n_a}, n_b/p\})$ por processador e $O(C \log p)$ rodadas de comunicação em que $O(n_a p^{1/C} + n_b)$ dados são transferidos de/para cada processador.*

Deve-se notar que para a resolução do problema LCS básico é necessário apenas conhecer D_G^0.

4.7 Obtenção da Maior Subseqüência Comum

Como já comentado no Capítulo 3, normalmente não se está interessado em obter *todos* os alinhamentos cujos valores foram determinados na resolução do ATS. Manter estruturas de dados que permitam a construção eficiente de qualquer alinhamento é custoso em termos de espaço. Assim, apenas os valores são obtidos e, caso necessário, um procedimento separado pode ser usado para obtenção de *um* alinhamento em particular.

No entanto, o algoritmo apresentado neste capítulo para o problema ALCS pode ser usado para resolução do problema LCS básico, uma vez que a complexidade deste algoritmo é satisfatória também para o LCS. Em outras palavras, ele pode ser aplicado em situações em que um alinhamento específico (mais exatamente, o da cadeia A com a cadeia B completa) é desejado.

4.7.1 Procedimento Básico

A seguir veremos extensões do algoritmo CGM para o ALCS que permitem a obtenção da maior subseqüência comum entre A e *qualquer* subcadeia de B em tempo $O(n_b \log p + n_a \log n_a)$ e $O(\log p)$ rodadas de comunicação.

Um procedimento recursivo é utilizado, decompondo-se o problema em GDAGs cada vez menores. Na verdade, as etapas de união são seguidas "ao contrário", utilizando dados coletados durante estas etapas.

Numa certa etapa de união, dois GDAGs S e I são unidos para determinar um GDAG U. Na fase de determinação do alinhamento, é preciso encontrar o melhor caminho entre $T_U(i) = T_S(i)$ e $F_U(j) = F_I(j)$. Encontra-se primeiro o vértice de $F_S = T_I$ que pertence a este caminho, digamos $F_S(x) = T_I(x)$. Depois procura-se o melhor caminho entre $T_S(i)$ e $F_S(x)$ e o melhor caminho entre $T_I(x)$ e $F_I(j)$, o que é feito recursivamente. Esta recursão pára quando os GDAGs básicos

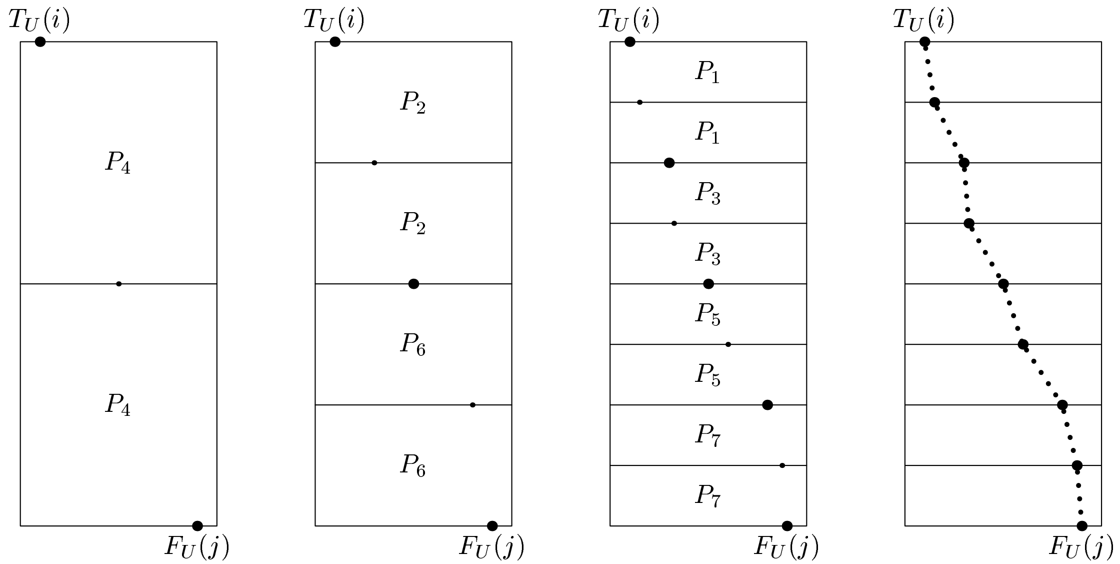

Figura 4.11: Obtenção do caminho entre $T_U(i)$ e $F_U(j)$. A cada etapa, os dados obtidos nas etapas de união são usados na ordem inversa para obter os pontos de cruzamento entre GDAGs. Na última etapa, os trechos de caminho nos GDAGs básicos são determinados. No interior de cada GDAG está indicado o processador que contém os dados relativos ao GDAG.

(que não foram produtos de um processo de união) são atingidos. Teremos então os p GDAGs que foram usados no início do algoritmo. Os caminhos nestes GDAGs podem ser determinados pelos p processadores em paralelo, em tempo $O(n_a n_b/p)$ e espaço $O(n_a/p + n_b)$ por processador, como mostrado na Seção 2.1.2. O caminho completo é formado pela união dos caminhos nos GDAGs básicos. A Figura 4.11 ilustra este processo.

A Figura 4.11 também mostra qual processador irá conter os dados de cada GDAG. Cada processador mantém os dados de no máximo dois GDAGs, vizinhos em alguma etapa de união. Para um certo processador P_t, os GDAGs que ele armazena são gerados na etapa de união $w(t)$ (ver Definição 3.2 na página 40). Se $w(t) = 0$ os GDAGs são básicos.

Cada processador usa estes dados na determinação de um dos vértices no caminho procurado. Esta distribuição permite que a determinação do caminho envolva pouquíssima comunicação.

4.7.2 Determinação dos Vértices Intermediários

Seja U um GDAG $(2m + 1) \times (n + 1)$ resultante da união de dois GDAGs $(m + 1) \times (n + 1)$ S e I, conforme descrito anteriormente. As estruturas de dados usadas para unir estes GDAGs são os vetores D_S^0, V_S, D_I^0 e V_I. Estas estruturas podem ser mantidas em memória após a etapa de união, pois não ocupam muito espaço. Por outro lado, a estrutura compacta da Seção 4.3 é grande demais para ser mantida depois que terminar a etapa de união em que ela foi criada e usada.

Para obter o caminho entre $T_U(i)$ e $F_U(j)$ é preciso usar dois vetores: D_S^i, que dá informações sobre os caminhos de $T_U(i)$ até a linha comum $F_U = T_I$, e $D_{I^{rev}}^j$, que dá informações sobre os caminhos da linha comum até $F_I(j)$. Este último vetor pode ser visto como o equivalente a D_I^j, se todos os arcos de I fossem *revertidos*. Sua definição é dada a seguir:

Definição 4.11 (Vetor $D_{I^{rev}}^j$) *Seja I um GDAG $(m+1) \times (n+1)$ para o problema ALCS. Para $0 \leq j \leq m$, $D_{I^{rev}}^j(0) = j$ e para $1 \leq k \leq n$, $D_{I^{rev}}^j(k)$ indica o valor i tal que $C_I(i,j) = k$ e $C_I(i+1,j) = k-1$. Se não houver tal valor então $D_{I^{rev}}(i,j) = -\infty$.*

É conveniente comparar as definições 4.2 (página 60) e 4.11 neste ponto. As duas definições são muito semelhantes, com papéis inversos para as variáveis i e j. Lembramos que $C_I(i,j)$ indica o peso do maior caminho entre $T_I(i)$ e $F_I(j)$. Note também que os elementos de $D_{I^{rev}}^j$ estão ordem *decrescente*.

Para uma melhor compreensão, a Tabela 4.7 mostra $D_{I^{rev}}^j$ para vários valores de j, usando no lugar de I o GDAG G da Figura 4.1. Os dados de C_I (no caso C_G) para o mesmo GDAG são mostrados na Tabela 4.6 (é a mesma tabela da página 62).

As seguintes observações são facilmente demonstráveis, decorrendo diretamente das definições:

Observação 4.3 *Para um GDAG G qualquer, dados dois vértices $T_G(i)$ e $F_G(j)$, o peso do maior caminho entre estes dois vértices ($C_G(i,j)$) é igual a:*

1. *o número de componentes de D_G^i que pertencem ao intervalo $]i,j]$.*
2. *o número de componentes de $D_{G^{rev}}^j$ que pertencem ao intervalo $[i,j[$.*

Estas observações reforçam a idéia de "simetria" que existe entre D_G e $D_{G^{rev}}$. A seguir, veremos como obter os vetores D_S^i e $D_{I^{rev}}^j$.

A determinação de $D_{I^{rev}}^j$ pode ser feita a partir de V_I. D_I^0 não é necessário. A Tabela 4.8 mostra V_I (no caso, V_G) na mesma página que $D_{I^{rev}}$ para facilitar a compreensão do seguinte lema:

Lema 4.9 *Para todo $i' < j$, i' é componente de $D_{I^{rev}}^j$ se e somente se $V_I(i'+1) > j$.*

Prova. Se $i' < j$ e $V_I(i'+1) > j$, isto significa, pela Definição 4.3, que não há nenhum componente de $D_I^{i'+1}$ que não esteja em $D_I^{i'}$ e seja menor que ou igual a j. No entanto, o componente i' pertence a $D_I^{i'}$ mas não a $D_I^{i'+1}$. Pela Observação 4.3(2) concluímos que $C_I(i',j) = C_I(i'+1,j)+1$. Pela Definição 4.11, i' é componente de $D_{I^{rev}}^j$.

Por outro lado, se $i' < j$ e i' é componente de $D_{I^{rev}}^j$, pela Definição 4.11 temos $C_I(i',j) = C_I(i'+1,j)+1$. Pela Observação 4.3(1), o número de componentes do intervalo $]i',j]$ em $D_I(i')$ deve ser menor do que em $D_I(i'+1)$. Isto só pode ocorrer se $V_I(i'+1) > j$ (o dado inserido em $D_I(i'+1)$ estiver fora do intervalo $]i',j]$). □

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

Tabela 4.6: C_G referente ao GDAG da Figura 4.1(reprise).

	k								
	0	1	2	3	4	5	6	7	8
$D_{G^{rev}}(13,k)$	13	12	11	10	9	6	5	3	0
$D_{G^{rev}}(12,k)$	12	11	10	9	6	5	3	1	0
$D_{G^{rev}}(11,k)$	11	10	9	8	6	5	3	1	0
$D_{G^{rev}}(10,k)$	10	9	8	6	5	3	2	1	0
$D_{G^{rev}}(9,k)$	9	8	7	6	5	3	2	1	0
$D_{G^{rev}}(8,k)$	8	7	6	5	3	2	1	0	$-\infty$
$D_{G^{rev}}(7,k)$	7	6	5	3	2	1	0	$-\infty$	$-\infty$
$D_{G^{rev}}(6,k)$	6	5	4	3	2	1	0	$-\infty$	$-\infty$
$D_{G^{rev}}(5,k)$	5	4	3	2	1	0	$-\infty$	$-\infty$	$-\infty$
$D_{G^{rev}}(4,k)$	4	3	2	1	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$D_{G^{rev}}(3,k)$	3	2	1	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$D_{G^{rev}}(2,k)$	2	1	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$D_{G^{rev}}(1,k)$	1	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$D_{G^{rev}}(0,k)$	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$

Tabela 4.7: $D_{G^{rev}}$ referente ao GDAG da Figura 4.1 (linhas em ordem inversa).

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

Tabela 4.8: V_G referente ao GDAG da Figura 4.1 (reprise). Pode ser usado na determinação de uma linha de $D_{G^{rev}}$.

Por este lema, temos uma maneira simples de construir D^j_{Irev} em tempo $O(n)$. Basta fazer $D^j_{Irev}(0) = j$ e verificar o vetor V_I do fim para o início, inserindo em D^j_{Irev} todo i' tal que $V_I(i'+1) > j$.

Para construir D^i_S é preciso inserir todos os dados de D^0_S e de V_S (até $V_S(i)$), desprezando os dados inferiores a i. Isto pode ser feito em tempo $O(m+n)$, obtendo-se um vetor de tamanho $O(m)$ que pode então ser ordenado em tempo $O(m \log m)$.

Concluímos então que os dois vetores necessários podem ser construídos em tempo $O(n + m \log m)$, sem grande espaço adicional.

A seguir, procede-se com a procura pelo vértice da linha comum. Seja l_0 o número de componentes de D^j_{Irev} no intervalo $[i, j[$, que pode ser determinado em tempo $O(m)$ e é igual a $C_I(i, j)$ (Observação 4.3(2)).

Para encontrar o vértice que está mais à esquerda, os candidatos naturais são os indicados em D^i_S. O primeiro candidato é $F_S(i) = T_I(i)$, logo abaixo de $T_S(i)$, pois $D^i_S(0) = i$. O maior caminho passando por este vértice tem peso l_0. Para os candidatos seguintes, o caminho até eles a partir de $T_S(i)$ é cada vez maior, mas o caminho até $F_I(j)$ pode diminuir.

Sendo $Max(k)$ o peso total do maior caminho passando pelo candidato k (ou seja, pelo vértice $F_S(D^i_S(k))$), pela Observação 4.3 temos:

$$Max(k) = l_0 + k - \text{número de componentes de } D^j_{Irev} \text{ no intervalo } [i, D^i_S(k)[\ .$$

A determinação destes valores, para todos os candidatos, pode ser feita em tempo total $O(m)$. A Figura 4.12 ilustra o processo. Determinando o maior destes valores, temos o vértice procurado.

Assim sendo, a partir dos dados de D^0_S, V_S e V_I é possível, em tempo $O(n + m \log m)$, determinar o vértice intermediário do caminho entre $T_U(i)$ e $F_U(j)$.

4.7.3 Análise do Processo de Obtenção da Maior Subseqüência Comum

Ao todo são necessárias $\log p$ etapas para determinar os vértices intermediários. Em cada etapa, cada processador envia/recebe $O(1)$ dados (indicando vértices determinados na etapa anterior).

Numa etapa em que os GDAGs têm dimensão $(m+1)\times(n+1)$, o tempo gasto é $O(n+m \log m)$. A cada etapa, m é reduzido pela metade, valendo $n_a/2$ na primeira. $n = n_b$ em todas as etapas. Disto conclui-se que o tempo gasto em todas as etapas é $O(n_b \log p + n_a \log n_a)$.

Os trechos do caminho contidos em GDAGs básicos podem ser determinados em tempo $O(n_b + n_a/p)$ se os dados usados durante a execução do Algoritmo 4.1 (ALCS seqüencial) forem mantidos, ocupando espaço $O(n_a n_b/p)$. Isto está indicado no Teorema 4.1. No entanto, é mais interessante não manter estes dados, para poupar espaço. Quando o caminho entre dois vértices específicos for requerido, usa-se um algoritmo seqüencial para o LCS que encontre este caminho em tempo $O(n_a n_b/p)$ usando espaço linear (ver Seção 2.1.2).

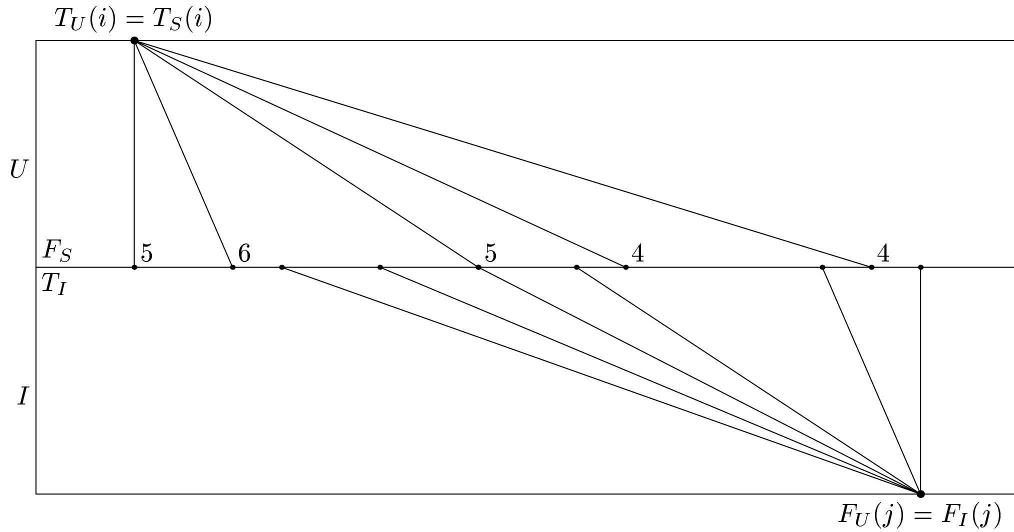

Figura 4.12: Obtenção do vértice intermediário do caminho entre $T_U(i)$ e $F_U(j)$. Os caminhos indicados em S vão de $T_U(i)$ até os vértices indicados por D_S^i. Os caminhos em I vão dos vértices indicados em D_{Irev}^j até $F_U(j)$. Junto a cada vértice indicado por D_S^i temos o peso total do maior caminho até $F_U(j)$ que passa por este vértice. Este peso envolve a contagem dos vértices indicados em D_{Irev}^j que estão à esquerda do vértice.

Cada processador determina então um trecho da maior subseqüência comum. Todos os trechos podem ser enviados a um único processador em uma única rodada de comunicação

Isto nos dá o seguinte resultado, que une as discussões desta seção com o resultado do Teorema 4.7:

Teorema 4.10 *Dadas duas cadeias A e B, respectivamente de comprimentos n_a e n_b, a determinação da maior subseqüência comum pode ser resolvido por $p < \sqrt{n_a}$ processadores em tempo $O\left(\frac{n_a n_b}{p}\right)$, espaço $O(n_b\sqrt{n_a})$ por processador e $O(C \log p)$ rodadas de comunicação em que $O(n_a p^{1/C} + n_b)$ dados são transferidos de/para cada processador.*

Capítulo 5

Conclusões

Problemas de alinhamento de cadeias estão entre os mais importantes problemas aplicáveis à biologia molecular. Atualmente, a quantidade de dados disponíveis é enorme, sendo necessário um grande poder de processamento para que estes dados possam ter sua utilidade explorada.

Técnicas de processamento paralelo têm sido exploradas há vários anos na resolução destes problemas. Porém, boa parte da literatura cobre técnicas específicas para o modelo PRAM, difíceis de adaptar para máquinas realísticas. Em particular, o problema LCS é abordado de diversas maneiras distintas em [10, 11, 31, 32], mas os algoritmos apresentados não são ótimos (com exceção de [32], base para os resultados do Capítulo 4) e dependem muito da existência de uma memória compartilhada.

No caso do paralelismo de granularidade grossa, uma técnica bastante explorada é a mostrada no Capítulo 2. Esta técnica é adequada em muitas aplicações práticas, mas o desempenho pode ser comprometido se o número de processadores for muito elevado e a rede de comunicação for lenta.

No caso específico do problema LCS, a técnica aqui apresentada é a que depende menos de comunicação entre processadores (pelo nosso conhecimento), por utilizar apenas $O(\log p)$ rodadas de comunicação. Esta característica deve contrabalançar a maior complexidade do algoritmo em sistemas com grande número de processadores, ou seja, esta técnica apresenta boa escalabilidade. Num futuro próximo, ela será implementada e comparada com técnicas mais simples. Parte dos resultados aqui apresentados foi submetida e aceita para publicação em anais no *17th International Parallel and Distributed Processing Symposium* (IPDPS 2003).

No caso do problema ATS (Capítulo 3), não há resultados publicados de nosso conhecimento que utilizem modelos de granularidade grossa. O algoritmo aqui apresentado foi publicado (SPAA 2002) [2], devendo também, num futuro próximo, ser implementado e avaliado na prática.

A experiência obtida no desenvolvimento destes dois algoritmos CGM indica uma abordagem promissora para futuros desenvolvimentos. Duas fontes de conhecimento prévio devem ser exploradas: os algoritmos seqüenciais e os algoritmos PRAM. Os algoritmos seqüenciais servem (não necessariamente) para a resolução de subproblemas, no caso de abordagens de divisão-e-

conquista. Os algoritmos PRAM expõem o paralelismo do problema de uma forma bastante ampla.

As principais dificuldades na adaptação de técnicas PRAM para o CGM estão na ausência de uma memória compartilhada (o que pode implicar em replicação de dados e aumento do espaço requerido) e na necessidade de reduzir o número de etapas de comunicação. As diferenças entre os algoritmos aqui apresentados e os algoritmos PRAM mais próximos ([32, 40]) ilustram como esta tarefa pode ser complexa. No entanto, é fácil adaptar um algoritmo CGM para uma máquina real de memória distribuída, o que justifica o uso do modelo.

Há uma enorme gama de problemas que ainda podem ser explorados. A partir dos resultados já obtidos, podemos seguir pelas seguintes linhas:

- Generalização de custos de edição, considerando custos diferenciados para remoção ou inserção de vários símbolos contíguos, por exemplo.
- Restrição de custos de edição, considerando apenas a distância de Levenshtein ou, de maneira mais genérica, custos de valor inteiro. O algoritmo aqui apresentado para o LCS deve fornecer uma base para estas pesquisas.
- Uso de técnicas apropriadas para alfabetos pequenos, como as apresentadas em [34, 35]. Estas técnicas levam a algoritmos seqüenciais sub-quadráticos para problemas de alinhamento, devido ao surgimento freqüente das mesmas subcadeias nas cadeias que estão sendo comparadas, mas no momento são consideradas de pouco uso prático.
- Estudo da Busca Aproximada por Cadeias [23, 25, 41], em que se procura uma cadeia dentro de um texto, tolerando-se uma quantidade máxima k (fixa) de erros.
- Estudo do Alinhamento Múltiplo, em que se procura a similaridade dentro de um *grupo* de cadeias, não apenas a comparação par a par. Este é um problema mais elaborado, que não foi explorado neste trabalho.
- Operações de edição mais sofisticadas, como inversão ou repetição de subcadeia.

As técnicas a serem usadas em cada um destes problemas podem ser bastante diferentes das que foram usadas nesta tese. Esta lista, que obviamente não está completa, serve para destacar o fato de que esta tese não esgota o assunto de que trata, mas abre caminho para novas pesquisas na área.

Apêndice A

Cálculos de Complexidade

Faremos um cálculo mais preciso do tempo de execução do Algoritmo 3.2, considerando aqui os trechos efetivamente usados de $meio_U$ em cada varredura. Este cálculo foi simplificado no Capítulo 3 para demonstração de complexidade assintótica, mas o cálculo mais preciso é necessário para demonstrar a qualidade do algoritmo paralelo da Seção 3.4. Mais exatamente, este cálculo é necessário para que se estabeleça uma relação entre o custo do maior subproblema possível e o custo total de todos os subproblemas, o que indica quão bom é o escalonamento de subproblemas para os processadores.

Antes, definimos funções que irão facilitar a apresentação. As seguintes funções já foram apresentadas anteriormente:

$$w : \mathbb{N}^+ \to \mathbb{N},\ w(n) = \max\{x \mid n \equiv 0 \pmod{2^x}\}$$
$$W : \mathbb{N}^+ \to \mathbb{N},\ W(n) = \textstyle\sum_{i=1}^{n} w(i)$$

É importante notar que todos os valores de i entre 1 e n tais que $w(i) \geq x$ são múltiplos de 2^x e os valores tais que $w(i) > x$ são múltiplos de 2^{x+1}. Disto decorre a afirmação a seguir

Afirmação A.1 *Os valores de i entre 1 e n tais que $w(i) = x$ formam uma progressão aritmética de termo geral $2^x(1+2u)$, $0 \leq u \leq \left\lfloor \frac{n-2^x}{2^{x+1}} \right\rfloor$.*

Definição A.1 (funções W_2, W_3 e W_4)

$$W_2 : \mathbb{N}^+ \to \mathbb{N},\ W_2(n) = \sum_{i=1}^{n} w(i)i$$

$$W_3 : \mathbb{N}^+ \to \mathbb{N},\ W_3(n) = \sum_{i=1}^{n} \frac{1}{2^{w(i)}}$$

$$W_4 : \mathbb{N}^+ \to \mathbb{N},\ W_4(n) = \sum_{i=1}^{n} \frac{i}{2^{w(i)}}$$

Pelo Lema 3.2 sabemos que $W(n) = n + O(\log n)$[1]. Os lemas a seguir determinam o comportamento assintótico de W_2, W_3 e W_4.

Lema A.1 $W_2(n) = \frac{n^2}{2} + O(n \log n)$

Prova. Como feito na demostração do Lema 3.2, usamos a notação de Iverson para escrever $w(i) = \sum_{j=1}^{\lfloor \log n \rfloor} [i \equiv 0 \pmod{2^j}]$ e portanto

$$\begin{aligned} W_2(n) = & \sum_{i=1}^{n} \left(\sum_{j=1}^{\lfloor \log n \rfloor} [i \equiv 0 \pmod{2^j}] \right) i = \sum_{j=1}^{\lfloor \log n \rfloor} \sum_{i=1}^{n} [i \equiv 0 \pmod{2^j}] i = \\ & \sum_{j=1}^{\lfloor \log n \rfloor} \sum_{k=1}^{\left\lfloor \frac{n}{2^j} \right\rfloor} k 2^j = \sum_{j=1}^{\lfloor \log n \rfloor} 2^j \frac{\left\lfloor \frac{n}{2^j} \right\rfloor \left\lfloor 1 + \frac{n}{2^j} \right\rfloor}{2} \end{aligned} \tag{A.1}$$

A partir de A.1 podemos deduzir os limites a seguir.

$$\begin{aligned} W_2(n) & \geq \sum_{j=1}^{\lfloor \log n \rfloor} 2^j \frac{\left(\frac{n}{2^j} - 1 \right) \frac{n}{2^j}}{2} = \frac{n}{2} \sum_{j=1}^{\lfloor \log n \rfloor} \left(\frac{n}{2^j} - 1 \right) = \frac{n}{2} \left(n - \frac{n}{2^{\lfloor \log n \rfloor}} - \lfloor \log n \rfloor \right) \\ & \geq \frac{n^2}{2} - n - \frac{n}{2} \lfloor \log n \rfloor \\ W_2(n) & \leq \sum_{j=1}^{\lfloor \log n \rfloor} 2^j \frac{\frac{n}{2^j} \left(\frac{n}{2^j} - 1 \right)}{2} = \frac{n}{2} \sum_{j=1}^{\lfloor \log n \rfloor} \left(\frac{n}{2^j} + 1 \right) = \frac{n}{2} \left(n - \frac{n}{2^{\lfloor \log n \rfloor}} + \lfloor \log n \rfloor \right) \\ & \leq \frac{n^2}{2} - \frac{n}{2} + \frac{n}{2} \lfloor \log n \rfloor \end{aligned}$$

Assim $W_2(n) = \frac{n^2}{2} + O(n \log n)$ □

Lema A.2 $W_3(n) = \frac{2n}{3} + O(1)$

Prova. Usando novamente a notação de Iverson, $[w(i) = x]$ é 1 se $w(i) = x$, 0 caso contrário. Assim sendo

$$\begin{aligned} W_3(n) = & \sum_{i=1}^{n} \frac{1}{2^{w(i)}} = \sum_{x=0}^{\lfloor \log n \rfloor} \left(\frac{1}{2^x} \sum_{i=1}^{n} [w(i) = x] \right) = \sum_{x=0}^{\lfloor \log n \rfloor} \frac{1}{2^x} \left(\frac{n}{2^{x+1}} + O(1) \right) = \\ & \left(\sum_{x=0}^{\lfloor \log n \rfloor} \frac{n}{2^{x+1}} \right) + O(1) = \frac{n}{2} \cdot \frac{1 - 4^{-\lfloor \log n \rfloor}}{\frac{3}{4}} + O(1) = \frac{2n}{3} \left(1 - \frac{1}{n^2} \right) + O(1) = \\ & \frac{2n}{3} + O(1) \end{aligned}$$

[1] Usamos aqui o seguinte "abuso notacional" [21]: $f(n) = g(n) + O(h(n))$ se e somente se $|f(n) - g(n)| = O(h(n))$.

□

Lema A.3 $W_4(n) = \frac{n^2}{3} + O(n)$

Prova.

$$W_4(n) = \sum_{i=1}^{n} \frac{i}{2^{w(i)}} = \sum_{x=0}^{\lfloor \log n \rfloor} \left(\frac{1}{2^x} \sum_{i=1}^{n} [w(i) = x] \cdot i \right) \tag{A.2}$$

Usando a Afirmação A.1 podemos calcular a somatória interna de A.2 tratando-a como a soma dos termos de uma progressão aritmética.

$$\sum_{i=1}^{n} [w(i) = x] \cdot i = 2^x \sum_{u=0}^{\left\lfloor \frac{n-2^x}{2^{x+1}} \right\rfloor} (1 + 2u) = 2^x \cdot \frac{1}{2} \left\lfloor \frac{n+2^x}{2^{x+1}} \right\rfloor \left(2 + 2 \left\lfloor \frac{n-2^x}{2^{x+1}} \right\rfloor \right)$$

Manipulando esta expressão, chegamos às seguintes desigualdades:

$$\frac{n^2}{2^{x+2}} - \frac{n}{2} + 2^{x-2} \leq \sum_{i=1}^{n} [w(i) = x] \cdot i \leq \frac{n^2}{2^{x+2}} + \frac{n}{2} + 2^{x-2} \tag{A.3}$$

Com A.3 aplicada a A.2 podemos encontrar um limite inferior e um superior para $W_4(n)$. Operando sobre o limite inferior:

$$\begin{aligned} W_4(n) = \quad & \geq \sum_{x=0}^{\lfloor \log n \rfloor} \frac{1}{2^x} \left(\frac{n^2}{2^{x+2}} - \frac{n}{2} + 2^{x-2} \right) = \sum_{x=0}^{\lfloor \log n \rfloor} \left(\frac{n^2}{2^{2x+2}} - \frac{n}{2^{x+1}} + \frac{1}{4} \right) = \\ & \frac{n^2}{4} \cdot \frac{1 - 4^{-\lfloor \log n \rfloor}}{\frac{3}{4}} + O(n) = \frac{n^2}{3} + O(n) \end{aligned}$$

Para o limite inferior para $W_4(n)$ obtemos resultados semelhantes, o que completa a demonstração do lema. □

A estimativa para o Algoritmo 3.2 dentro do contexto apresentado na Seção 3.2 é dada abaixo. Como no caso da demonstração do Teorema 3.3, estamos ignorando as varreduras feitas com os extremos de $orig_S$ e $dest_I$. Desta vez, no entanto, estamos separando $orig_S$ e $dest_I$ nas partes referentes a E_S, T_S, F_I e D_I.

$$\begin{aligned} T(t,k) = \quad & \underbrace{\sum_{i=1}^{k-1} v_d(i, t-k+i+1, i+1)}_{TF_I(t,k)} + \underbrace{\sum_{i=k}^{t-2} v_d(i,t,k)}_{TD_I(t,k)} + \\ & \underbrace{\sum_{i=1}^{t-k} v_o(i,t,k)}_{TE_S(t,k)} + \underbrace{\sum_{i=t-k+1}^{t-2} v_o(i, 2t-k-i+1, t-i)}_{TT_S(t,k)} \end{aligned} \tag{A.4}$$

A principal diferença deste cálculo para o realizado anteriormente é no uso das funções v_d e v_o. Para varreduras lideradas por vértices de T_S foram considerados apenas vértices alcançáveis de $dest_I$, o que se reflete no segundo parâmetro de v_o, e os trechos de $meio_U$ que podem ser usados, o que se reflete no terceiro parâmetro de v_o. Considerações semelhantes foram feitas para as varreduras lideradas por vértices de F_I. Expandindo as funções v_d e v_o segundo a Definição 3.3 em cada termo de A.4, já desprezando os termos de ordem inferior ($o(k^2)$), temos:

$$\begin{aligned}
TF_I(t,k) = & \sum_{i=1}^{k-1}\left(w(i)i + 2(t-k+i) - \frac{t-k+i}{2^{w(i)-1}}\right) + o(k^2) = \\
& W_2(k) + 2tk - 2k^2 + k^2 - 2(t-k)W_3(k) - 2W_4(k) + o(k^2) = \\
& \frac{2tk}{3} - \frac{k^2}{6} + o(k^2)
\end{aligned}$$

$$\begin{aligned}
TD_I(t,k) = & \sum_{i=k}^{t-2}\left(w(i)k + 2t - \frac{t}{2^{w(i)-1}}\right) + o(k^2) = \\
& (W(t) - W(k))k + 2t^2 - 2tk - 2t(W_3(t) - W_3(k)) + o(k^2) = \\
& \frac{2t^2}{3} + \frac{tk}{3} - k^2 + o(k^2)
\end{aligned}$$

$$\begin{aligned}
TE_S(t,k) = & \sum_{i=1}^{t-k}\left(w(i)k + 2t - \frac{t}{2^{w(i)}}\right) = \\
& W(t-k)k + 2t(t-k) - W_3(t-k)t + o(k^2) = \\
& \frac{4t^2}{3} - \frac{tk}{3} - k^2 + o(k^2)
\end{aligned}$$

$$\begin{aligned}
TT_S(t,k) = & \sum_{i=t-k+1}^{t-2}\left(w(i)(t-i) + 2(2t-k-i) - \frac{2t-k-i}{2^{w(i)}}\right) = \\
& t(W(t) - W(t-k)) - (W_2(t) - W_2(t-k)) + 2k(2t-k) - k(2t-k) \\
& \quad -(2t-k)(W_3(t) - W_3(t-k)) + W_4(t) - W_4(t-k) + o(k^2) = \\
& \frac{4tk}{3} - \frac{k^2}{6} + o(k^2)
\end{aligned}$$

Reunindo os resultados anteriores à equação A.4 temos

$$T(t,k) = 2t^2 + 2tk - \frac{k^2}{3} + o(k^2) \tag{A.5}$$

Podemos então provar o seguinte lema:

Lema A.4 *A razão entre o maior tempo de execução possível para um subproblema que surge da decomposição descrita na Seção 3.4.2 e o tempo total de todos os subproblemas é inferior a* $\frac{1}{2q}$

Prova. O tempo total de todos os subproblemas pode ser estimado pela equação A.5. O maior tempo possível para um subproblema é o que ocorre quando $t' \approx \frac{t}{2q}$ vértices de E_S e t' vértices de D_I estão envolvidos, sendo que todos os vértices de $meio_U$ devem ser explorados (tal subproblema pode de fato aparecer). A seguir temos o cálculo de uma estimativa de tempo, boa a menos de uma constante multiplicativa que é essencialmente a mesma envolvida no cálculo do tempo total.

$$
\begin{aligned}
TSub_{max}(t,k,q) = \quad & \sum_{i=1}^{t'-1} v_d(i,t',k) + \sum_{i=1}^{t'-1} v_o(i,t',k) = \\
& \sum_{i=1}^{t'-1}\left(w(i)k + 2t' - \frac{t'}{2^{w(i)-1}}\right) + \sum_{i=1}^{t'-1}\left(w(i)k + 2t' - \frac{t'}{2^{w(i)}}\right) + o\left(\frac{k^2}{q}\right) = \\
& \sum_{i=1}^{t'-1}\left(2w(i)k + 4t' - \frac{3t'}{2^{w(i)}}\right) = 2W(t')k + 4(t')^2 - 3W_3(t')t' + o\left(\frac{k^2}{q}\right) = \\
& 2kt' + 2(t')^2 + o\left(\frac{k^2}{q}\right) = \frac{kt}{q} + \frac{t^2}{2q^2} + o\left(\frac{k^2}{q}\right)
\end{aligned}
$$

Com isto, usando o fato de que $k \leq t$, temos as seguintes desigualdades:

$$
\begin{aligned}
\tfrac{T(t,k)}{2q} = \quad & \frac{2t^2 + 2tk - \frac{k^2}{3}}{2q} + o\left(\frac{k^2}{q}\right) = \frac{t^2}{q} + \frac{tk}{q} - \frac{k^2}{6q} + o\left(\frac{k^2}{q}\right) \geq \\
& \frac{t^2}{q} - \frac{t^2}{6q} + \frac{tk}{q} + o\left(\frac{k^2}{q}\right) = \frac{5t^2}{6q} + \frac{tk}{q} + o\left(\frac{k^2}{q}\right) > TSub_{max}(t,k,q)
\end{aligned}
$$

□

Com relação ao espaço necessário para cada subproblema, podemos provar o seguinte lema:

Lema A.5 *A razão entre o maior espaço possível necessário para um subproblema que surge da decomposição descrita na Seção 3.4.2 e o espaço total para todos os subproblemas é inferior a* $\frac{5}{6q}$

Prova. Chamamos $E(t,k)$ o espaço necessário para um subproblema para certos valores de t e k. $ESub_{max}(t,k,q)$ é o maior espaço possível para um subproblema.

O espaço necessário para armazenar AL'_S e AL'_I é $2tk - k^2$ e o necessário para armazenar AL'_U é $t^2 - \frac{k^2}{2}$, totalizando $E(t,k) = 2tk + t^2 - \frac{3k^2}{2}$ (veja Figura 3.3). O maior subproblema possível usaria espaço $ESub_{max}(t,k,q) = 2(t')k + (t')^2 = \frac{tk}{q} + \frac{t^2}{4q^2}$ para as partes necessárias de AL'_S, AL'_I e AL'_U, correspondendo a um bloco $t' \times t'$ de AL'_U e uma faixa $t' \times k$ de cada matriz AL'_S e AL'_I. Estes valores são estimativas boas a menos de uma *mesma* constante multiplicativa.

Assim sendo,

$$\frac{5}{6q} \cdot E(t,k) = \frac{20kt + 10t^2 - 15k^2}{12q} = \frac{kt}{q} + \frac{8kt + 10t^2 - 15k^2}{12q}$$

Como $1 \leq k \leq t$, o segundo termo é mínimo para $k = t$. Considerando também que $q \geq 1$ temos

$$\frac{5}{6q} \cdot E(t,k) \geq \frac{kt}{q} + \frac{8t^2 + 10t^2 - 15t^2}{12q} = \frac{kt}{q} + \frac{t^2}{4q} \geq \frac{kt}{q} + \frac{t^2}{4q^2} = ESub_{max}(t,k,q)$$

□

Referências Bibliográficas

[1] Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter Shor, and Robert Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987.

[2] 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 of the 14th Symposium on Parallel Algorithms and Architectures (ACM-SPAA)*, pages 275–281. ACM Press, 2002.

[3] C. E. R. Alves, E. N. Cáceres, F. Dehne, and S. W. Song. A parameterized parallel algorithm for efficient biological sequence comparison. Relatório Técnico RT–MAC–2002–06, Instituto de Matemática e Estatística, USP, Agosto 2002.

[4] C. E. R. Alves, E. N. Cáceres, F. Dehne, and S. W. Song. A CGM/BSP parallel similarity algorithm. In *Proceedings of the I Brazilian Workshop on Bioinformatics*, pages 1–8, october 2002.

[5] 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 of the 17th International Parallel and Distributed Processing Symposium (IEEE-IPDPS)*. IEEE Computer Society Press, 2003.

[6] A. Apostolico, M. J. Atallah, L. L. Larmore, and S. Macfaddin. Efficient parallel algorithms for string editing and related problems. *SIAM J. Comput.*, 19(5):968–988, 1990.

[7] A. Apostolico and R. Giancarlo. Sequence alignment in molecular biology. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 47:85–115, 1999.

[8] A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2:315–336, 1987.

[9] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economic construction of the transitive closure of a directed graph. *Dokl. Acad. Nauk. SSSR*, 194:487–488, 1970.

[10] K. Nandau Babu and Sanjeev Saxena. Parallel algorithms for the longest common subsequence problem. In *Proceedings of the International Conference on High Performance Computing*, 1997.

[11] Joan Boyar and Kim S. Larsen. Faster parallel computation of edit distance. Technical Report PP–1994–11, IMADA - Dept. of Mathematics and Computer Science, University of Southern Denmark, 1994.

[12] E. N. Cáceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. W. Song. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Proceedings ICALP'97 - 24th International Colloquium on Automata, Languages, and Programming*, volume 1256 of *Lecture Notes in Computer Science*, pages 390–400. Springer Verlag, 1997.

[13] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. 4th ACM SIGPLAN Symp. On Principles and Practice of Parallel Programming*, pages 1–12, 1993.

[14] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. In *Proc. 9th Annual ACM Symp. Comput. Geom.*, pages 298–307, 1993.

[15] F. Dehne (Ed.). Coarse grained parallel algorithms. *Special Issue of Algorithmica*, 24(3/4):173–176, 1999.

[16] E. W. Edmiston, N. G. Core, J. H. Saltz, and R. M. Smith. Parallel processing of biological sequence comparison algorithms. *International Journal of Parallel Programming*, 17(3):259–275, 1988.

[17] M. Gengler. An introduction to parallel dynamic programming. *Lecture Notes in Computer Science*, 1054:87–114, 1996.

[18] Alexandros V. Gerbessiotis and Leslie G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.

[19] P. B. Gibbons. A more practical PRAM model. In *Proceedings of the 1st Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 158–168. ACM Press, 1989.

[20] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17:416–426, 1969.

[21] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Matemática Concreta, Fundamentos para a Ciência da Computação.* LTC Editora, second edition, 1995.

[22] Anthony J. F. Griffiths, Jeffrey H. Miller, David T. Suzuki, Richard C. Lewontin, and William M. Gelbart. *An Introduction to Genetic Analysis.* W. H. Freeman and Company, sixth edition, 1996.

[23] D. Gusfield. *Algorithms in Strings, Trees and Sequences. Computer Science and Computational Biology.* Cambridge University Press, 1997.

[24] Leslie A. Hall. Approximation algorithms for scheduling. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, chapter 1, pages 1–45. PWS Publishing Company, 1995.

[25] P.A. Hall and G.R. Dowling. Approximate string matching. *Comput. Surveys*, 12(4):381–402, 1980.

[26] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communication of ACM*, 18:341–343, 1975.

[27] Xiaoqiu Huang. A space-efficient parallel sequence comparison algorithm for a message-passing multiprocessor. *International Journal of Parallel Programming*, 18(3):223–239, 1989.

[28] James W. Hunt and Thomas G. Szymansky. A fast algorithm for computing longest common subsequences. *Comm. ACM*, 20(5):350–353, May 1977.

[29] Gad M. Landau and Michal Ziv-Ukelson. On the common substring alignment problem. *Journal of Algorithms*, 41:338–359, 2001.

[30] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Phys. Dokl.*, 10:707–710, 1966.

[31] Mi Lu. Parallel computation of longest-common-subsequences. *Lecture Notes on Computer Science - International Conference on Computing and Information*, 468:385–394, 1990.

[32] Mi Lu and Hua Lin. Parallel algorithms for the longest common subsequence problem. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):835–848, 1994.

[33] M. Maes. On a cyclic string-to-string correction problem. *Information Processing Letters*, 35:73–78, 1990.

[34] W. J. Masek and M. S. Paterson. A faster algorithm for computing string edit distances. *J. Comp. Sys, Sci*, 20:18–31, 1980.

[35] W. J. Masek and M. S. Paterson. How to compute string-edit distances quickly. In D. Sankoff and J. Kruskal, editors, *Time warps, string edits, and macromolecules: the theory and practice of sequence comparison*, pages 337–349. Addison Wesley, 1983.

[36] H. Mongelli. *Algoritmos CGM para Busca Uni e Bidimensional de Padrões com e sem Escala.* PhD thesis, Instituto de Matemática e Estatística - USP - São Paulo/SP - Brasil, Abril 2000.

[37] Pavel A. Pevzner. *Computational Molecular Biology - An Algorithmic Approach.* The MIT Press, 2000.

[38] Sanjay Ranka and Sartaj Sahni. String editing on an SIMD hypercube multicomputer. *Journal of Parallel and Distributed Computing*, 9:411–418, 1990.

[39] Claus Rick. New algorithms for the longest common subsequence problem. Technical Report 85123–CS, Institut für Informatik, Universität Bonn, 1994.

[40] 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.

[41] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology.* PWS Publishing Company, 1997.

[42] Leslie G. Valiant. A bridging model for parallel computation. *Communication of the ACM*, 33(8):103–111, 1990.

[43] Leslie G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 943–972. Elsevier/MIT Press, 1990.

[44] S. Wu, U. Manber, G. Myers, and W. Miller. An $O(NP)$ sequence comparison algorithm. *Information Processing Letters*, 35:317–323, 1990.

[45] Tien K. Yap and Robert L. Martino. Parallel computation in biological sequence analysis. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):1–12, 1998.