**Algoritmos Paralelos para
Construção da Árvore dos Sufixos
e suas Aplicações**

Fábio Henrique Viduani Martinez

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

Área de Concentração: **Ciência da Computação**
Orientador: **Prof. Dr. Siang Wun Song**

Durante a elaboração deste trabalho o autor recebeu apoio financeiro do CNPq

-São Paulo, maio de 1996-

Algoritmos Paralelos para
Construção da Árvore dos Sufixos
e suas Aplicações

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Fábio Henrique Viduani Martinez
e aprovada pela comissão julgadora.

São Paulo, 31 de maio de 1996.

Banca examinadora:

- Prof. Dr. Siang Wun Song (orientador) - IME-USP
- Prof. Dr. José Augusto Ramos Soares - IME-USP
- Prof. Dr. João Carlos Setubal - IMECC-UNICAMP

aos meus avós
Evaristo e Augusta
(in memoriam)

Agradecimentos

Ao Prof. Siang, por ter me aceito como orientado, pela paciência e dedicação na orientação e por todos os ensinamentos.

A minha querida Edna, pelo sofrimento em minhas derrotas e pela vibração nas conquistas, mesmo a mais de mil quilômetros de distância.

A minha família e, em especial, a meus pais e a minha irmã, pelo apoio e pela torcida constante.

Ao Prof. José Augusto, pela paciência e pelo incentivo em momentos cruciais.

Ao Marcelo Lauretto, companheiro de longa caminhada, por estar sempre presente. Ao Marco Aurélio, pela amizade incondicional. Ao Francisco, pela amizade e por ter conseguido conviver comigo por mais de dois anos no ap. do CRUSP. E aos amigos que fizeram este mestrado mais ameno: Alexandre, Ayumi, Cao, Carlos, Claus, Edson, Eduardo, Fátima, Flávio, Guillermo, Jair, Juaci, Lúcia e Renata. Espero ter lembrado de todos!

Resumo

A árvore dos sufixos é, notavelmente, uma das estruturas de dados mais estudadas em processamento de palavras. O principal objeto de estudo de nossa dissertação são os algoritmos paralelos para construção da árvore dos sufixos.

Enfatizamos nosso estudo no algoritmo paralelo proposto por Apostolico et. al em 1988, devido, especialmente, a sua simplicidade e também por ser o melhor dentre os primeiros algoritmos paralelos propostos para a construção da árvore dos sufixos. Apresentamos ainda uma visão geral de funcionamento de algoritmos paralelos mais recentes, onde o trabalho ótimo é a principal característica.

Além do estudo de algoritmos paralelos para construção da árvore dos sufixos, apresentamos algumas aplicações paralelas. Entendemos por aplicações paralelas o estudo de algoritmos paralelos eficientes que utilizam a árvore dos sufixos em um primeiro estágio de pré-processamento. Neste contexto, tentamos descrever de maneira detalhada especialmente o problema do casamento de padrões e um algoritmo paralelo para solucioná-lo.

Abstract

Suffix trees are notably among the most studied data structures in string processing. The main objective of our dissertation is to study parallel algorithms for the construction of suffix trees.

In our study we emphasize the parallel algorithm proposed by Apostolico et al. in 1988 due mainly to its simplicity and to being the best among the first parallel algorithms proposed for the construction of suffix trees. Also we present a general view on how some recent algorithms work. The main characteristic of these algorithms is the optimal work.

In addition to the study of parallel algorithms for the construction of suffix trees, we present some parallel applications. By parallel application we mean the study of efficient parallel algorithms that utilize the suffix tree in a first pre-processing stage. Under this context, we attempt to describe especially the pattern matching problem in a detailed way, as well as a parallel algorithm for its solution.

Índice

1	Introdução	1
2	Preliminares	5
2.1	Palavras, Alfabetos e Linguagens	5
2.2	Computação Paralela	6
2.2.1	Processamento Paralelo	7
2.2.2	Modelos de Computação Paralela	8
2.2.3	Estrutura Trabalho-Tempo de Apresentação	10
2.2.4	Otimidade	11
2.2.5	Simulação entre Modelos PRAM	12
2.2.6	P -completude	14
3	Árvore dos Sufixos	17
3.1	Definição Inicial	18
3.2	Formalização	19
3.3	Altura e Tamanho da Árvore dos Sufixos	27
4	Construção da Árvore dos Sufixos	31
4.1	Algoritmos Sequenciais	32
4.2	Algoritmos Paralelos	33
4.2.1	Algoritmo Landau-Schieber-Vishkin	34
4.2.2	Algoritmo AILSV	36
4.2.3	Algoritmo Sahinalp-Vishkin	62

4.2.4	Algoritmo Hariharan	66
5	Aplicações	71
5.1	Maior Subpalavra Repetida	72
5.2	Identificadores de Subpalavras	72
5.3	Casamento de Padrões <i>On-line</i>	74
5.4	Outras Aplicações	81
6	Considerações finais	89
A	Implementação do Algoritmo AILSV	93
A.1	Algoritmo Seqüencial	94
A.2	Exemplos	98
B	Lista de Símbolos	103
	Bibliografia	105

Capítulo 1

Introdução

Além de desempenhar um importante papel em processamento de palavras e em diversas aplicações, o projeto de algoritmos eficientes para construção da árvore dos sufixos apresenta grande interesse por si só. Nossa dissertação tem como principal objetivo o estudo dos algoritmos paralelos para a construção da árvore dos sufixos.

Diversos tipos de processamento de palavras podem ser expressos dentro do seguinte contexto. Dada uma palavra x , pré-processamos x de maneira a estabelecer uma estrutura de dados apropriada, que permite o processamento eficiente de consultas sobre subpalavras de x . Exemplos de consultas são o casamento de padrões, encontrar a maior subpalavra repetida da palavra x , encontrar o maior prefixo comum de duas subpalavras da palavra x , entre outras.

A árvore dos sufixos, associada a uma palavra x , é uma estrutura de dados que suporta o processamento dessas tarefas. Desde seu surgimento em 1973 [Wei73], a árvore dos sufixos foi se tornando uma importante ferramenta no processamento de palavras. Atualmente, existe um conjunto amplo e bem diversificado de aplicações que a utilizam, que inclui desde aplicações mais imediatas, como a busca da maior subpalavra repetida, até mesmo aplicações complexas e sofisticadas como a análise de seqüências genéticas para identificação de padrões biologicamente significativos do DNA.

A árvore dos sufixos está inserida dentro do contexto da busca digital. O método da busca digital utiliza a representação de um conjunto de chaves como sendo uma seqüência de símbolos e extrai as informações necessárias dessa representação em uma busca. A árvore *trie*¹ [Fre60], uma árvore de busca digital, é a base da definição da árvore dos sufixos. Quando uma árvore *trie* está associada ao conjunto de todos os sufixos de uma determinada palavra, essa árvore é também chamada de árvore dos sufixos esticada. Através da compactação de caminhos de vértices de grau de saída igual a 1 da árvore dos sufixos

¹Esse nome vem de recuperação da informação (*retrieval information*), e lê-se *try* como no inglês.

esticada, obtém-se a estrutura que será o objeto principal de nosso estudo, a árvore dos sufixos.

Algumas informações importantes sobre a árvore dos sufixos são a sua quantidade de vértices e a sua altura. Neste trabalho, conseguimos mostrar que a quantidade de vértices da árvore dos sufixos é no máximo linear no tamanho da palavra. No entanto, não conseguimos mostrar nenhum limitante razoável para a altura. Considerações probabilísticas relevantes sobre a altura de uma árvore dos sufixos associada a uma palavra aleatória são realizadas por Szpankowski e por Mahmoud, em diversos trabalhos [JS91, DSR92, Szp92, Szp93b, Mah92].

A motivação inicial para a criação da árvore dos sufixos foi o desenvolvimento de uma aplicação específica: o casamento de padrões. O problema do casamento de padrões consiste em encontrar uma ocorrência de uma palavra y , denominada padrão, de tamanho m , em uma palavra x , denominada texto, de tamanho n , com $m \leq n$. Existem vários algoritmos que solucionam esse problema. De um modo geral, esses algoritmos podem ser divididos em duas classes: os algoritmos que pré-processam o padrão e então realizam sua busca no texto e os algoritmos que pré-processam o texto, construindo uma estrutura apropriada, e então consultam essa estrutura para encontrar todas as ocorrências do padrão no texto.

O algoritmo de Weiner [Wei73] foi o primeiro algoritmo seqüencial proposto para a solução do problema de casamento de padrões através da utilização da árvore dos sufixos. O pré-processamento, isto é, a construção da árvore dos sufixos associada à palavra x de tamanho n , sobre o alfabeto Σ , tem complexidade de tempo $O(n \log |\Sigma|)$. Caso o tamanho do alfabeto Σ seja fixo, esse algoritmo tem complexidade de tempo linear. Outros algoritmos seqüenciais para a construção da árvore dos sufixos podem ser citados tais como o algoritmo de McCreight [McC76], o algoritmo de Kosaraju [Kos94] e, recentemente, os algoritmos de Andersson e Nilsson [AN95], de Kosaraju e Delcher [KD95] e de Ukkonen [Ukk95].

No estudo dos algoritmos para construção da árvore dos sufixos estamos especialmente interessados nos algoritmos paralelos desenvolvidos com esse objetivo. Os algoritmos estudados foram projetados no modelo PRAM CRCW. Podemos considerar, daqui por diante, que a palavra de entrada x tem tamanho n e foi obtida em um alfabeto Σ de tamanho constante.

Um trabalho pioneiro nessa área foi proposto por Landau e Vishkin em 1986. O algoritmo desenvolvido tem complexidade de tempo $O(\log n)$ e de trabalho $O(n^2)$.

Em seguida, Apostolico e Iliopoulos [AI87], simultaneamente a Landau, Schieber e Vishkin [LSV87] propuseram algoritmos equivalentes com complexidade de tempo $O(\log n)$, complexidade de trabalho $O(n \log n)$ e de espaço $O(n^2)$. No ano seguinte, Apostolico, Iliopoulos, Landau, Schieber e Vishkin [AIL⁺88] desenvolveram um algoritmo que reúne as principais vantagens dos algoritmos anteriores. No entanto, as complexida-

des desse novo algoritmo permanecem equivalentes. Um esquema para redução do espaço para $O(n^{1+\delta})$, para qualquer $0 < \delta \leq 1$, é também sugerido pelos autores. Este algoritmo será apresentado de modo detalhado neste trabalho.

É importante notar que existem diversos algoritmos seqüenciais para construção da árvore dos sufixos de complexidade de tempo linear, mas o número total de operações realizadas pelos algoritmos paralelos anteriores é proporcional a $n \log n$, independentemente do tamanho do alfabeto relacionado. Esses algoritmos estão baseados no procedimento clássico de atribuição de nomes de [KMR72]. Esse procedimento produz uma estrutura de dados extremamente útil para algoritmos paralelos que trabalham com palavras, como mostrado em [CR91]. No entanto, essa técnica impossibilita que o trabalho desempenhado pelo algoritmo seja linear devido, sucintamente, ao número de subpalavras, proporcional a $n \log n$, às quais o algoritmo deve atribuir nomes. Esse número teoricamente dita a complexidade de trabalho do algoritmo. O fator logarítmico se deve ao aumento das redundâncias entre as subpalavras, já que ocorrem sobreposições a medida que elas se tornam maiores. O principal problema é que não existe uma forma consistente de selecionar somente uma subpalavra em um conjunto de subpalavras sobrepostas.

Recentemente, Sahinalp e Vishkin [SV94] propuseram um algoritmo que quebra a simetria que ocorre no processo de atribuição de nomes, e assim reduziram o número de operações realizadas para linear. O algoritmo proposto tem complexidade de tempo $O(\log^2 n)$ e complexidade de trabalho $O(n)$. A complexidade de espaço continua sendo $O(n^{1+\delta})$, para qualquer $0 < \delta \leq 1$.

Em um outro trabalho recente, Hariharan [Har94] obteve o primeiro algoritmo paralelo de trabalho e espaço ótimo, que constrói a árvore dos sufixos em tempo $O(\log^4 n)$.

Farach, Hariharan e Muthukrishnan [FHM], em um trabalho ainda não publicado, propõem um algoritmo paralelo aleatorizado para construção da árvore dos sufixos, de complexidade de tempo $O(\log n)$, complexidade de trabalho $O(n)$ e complexidade de espaço $O(n)$.

Um resultado experimental surpreendente foi obtido por Vishkin [Vis94], sugerindo que a emulação seqüencial de um algoritmo paralelo tem a vantagem potencial de ser executado sobre uma máquina seqüencial mais rapidamente que um algoritmo seqüencial padrão para solucionar o mesmo problema. Em particular, uma comparação entre o algoritmo seqüencial de McCreight [McC76] e o algoritmo paralelo seqüencialmente emulado de Apostolico et. al [AIL⁺88] é realizada e, sob determinadas condições, os resultados experimentais obtidos demonstram que o segundo algoritmo é muitas ordens de magnitude mais rápido que o primeiro.

Aplicações para a construção paralela da árvore dos sufixos são constantemente propostas. Entendemos por aplicações o estudo de algoritmos paralelos eficientes para solução de problemas relacionados a processamento de palavras, que utilizam a construção da árvore dos sufixos em um primeiro estágio de pré-processamento.

As principais idéias de nosso estudo, que acabamos de apresentar, serão detalhadas nos capítulos seguintes de nossa dissertação e estão organizadas da seguinte maneira.

No Capítulo 2, apresentaremos um conjunto de conceitos básicos sobre palavras, bem como conceitos fundamentais sobre Computação Paralela, fixando o modelo paralelo em que trabalharemos e apresentando uma estrutura geral para projeto e análise de algoritmos paralelos. Ainda nesse capítulo, apresentaremos simulações entre as variantes desse modelo paralelo fixado, assim como uma breve introdução à classe de complexidade onde estão incluídos os algoritmos que estudaremos.

No Capítulo 3, além de um conjunto de definições e propriedades básicas relacionadas à árvore dos sufixos, apresentaremos ainda considerações importantes, sob o ponto de vista das suas aplicações, sobre a altura e o tamanho da árvore dos sufixos.

A construção da árvore dos sufixos é o assunto tratado no Capítulo 4. Inicialmente, fazemos uma síntese dos algoritmos sequenciais existentes para a construção da árvore dos sufixos. Em seguida, descrevemos os algoritmos paralelos em [LSV87, AIL⁺88, SV94, Har94]. O algoritmo proposto em [AIL⁺88] é descrito detalhadamente por nós na Estrutura Trabalho-Tempo de Apresentação de Algoritmos Paralelos. Aos algoritmos paralelos restantes, damos apenas uma visão geral de funcionamento.

Muitas são as aplicações da construção da árvore dos sufixos no projeto de algoritmos paralelos eficientes. No Capítulo 5, tentamos descrever diversas delas. Vamos descrever detalhadamente o problema de encontrar a maior subpalavra repetida de uma dada palavra, o problema de encontrar o identificador de uma subpalavra, dados um índice e uma palavra e o problema do casamento de padrões *on-line* (*on-line string matching*). Em seguida, aplicações mais sofisticadas são descritas, com menos profundidade: encontrar o menor ancestral comum, casamento aproximado de padrões (*approximate string matching*), construção do autômato dos sufixos minimal e do autômato dos fatores minimal, a generalização da árvore dos sufixos associada a matrizes quadradas e compressão de dados. Alguns desses problemas possuem aplicações em outras áreas, tais como o casamento aproximado de padrões e a generalização da árvore dos sufixos para matrizes quadradas, com aplicações em biologia molecular computacional e em processamento digital de imagens de baixo nível, respectivamente.

O Capítulo 6 contém as considerações finais de nossa dissertação.

A tentativa de uma implementação do algoritmo paralelo descrito em [AIL⁺88] é apresentada no Apêndice A.

Finalmente, uma lista de símbolos utilizados, com o significado sucinto de cada um deles, bem como sua primeira ocorrência neste trabalho, encontra-se no Apêndice B.

Capítulo 2

Preliminares

Apresentaremos neste capítulo um conjunto de conceitos básicos que serão frequentemente utilizados por todos os capítulos posteriores de nossa dissertação. Neste conjunto estão incluídos conceitos básicos tais como símbolos, palavras, alfabetos e linguagens e também conceitos fundamentais sobre Computação Paralela, fixando o modelo paralelo em que trabalharemos e apresentando uma estrutura geral para projeto e análise de algoritmos paralelos. Finalizando o capítulo, apresentaremos simulações entre as variantes desse modelo paralelo fixado, assim como uma breve introdução à classe de complexidade onde estão incluídos os algoritmos que estudaremos.

2.1 Palavras, Alfabetos e Linguagens

Apresentaremos aqui alguns conceitos básicos sobre palavras, alfabetos e linguagens extraídos do livro de Lewis e Papadimitriou [LP81].

Um **símbolo** é uma entidade abstrata que não será definida formalmente. Letras e dígitos são exemplos de símbolos frequentemente utilizados. Um **alfabeto**, denotado por Σ , é um conjunto finito de símbolos. Uma **palavra** sobre um alfabeto é uma sequência finita de símbolos justapostos do alfabeto. Por exemplo, a , b e c são símbolos de um alfabeto Σ e $abca$ é uma palavra sobre Σ . O **tamanho** de uma palavra w , denotado por $|w|$, é o número de símbolos que compõem a palavra. Por exemplo, $abca$ tem tamanho 4. A palavra vazia, denotada por ϵ , é a palavra de zero símbolos. Assim, $|\epsilon| = 0$. Duas palavras sobre o mesmo alfabeto podem ser combinadas para compor uma terceira através da operação de **concatenação**. A concatenação das palavras x e y , denotada por $x \cdot y$ ou simplesmente por xy , é a palavra x seguida da palavra y . Formalmente, $w = x \cdot y$ se e somente se $|w| = |x| + |y|$, $w_j = x_j$, para $j = 1, \dots, |x|$ e $w_{|x|+j} = y_j$, para $j = 1, \dots, |y|$. Por exemplo, $abc \cdot cba = abccba$. Claramente, $w \cdot \epsilon = \epsilon \cdot w = w$, para qualquer w . Além

disso, a operação de concatenação é associativa: $(xy)w = x(yw)$, para quaisquer x, y e w .

Uma **subpalavra** é uma seqüência contígua qualquer de símbolos de uma palavra, de qualquer tamanho. Por exemplo, $\epsilon, a, b, c, d, ab, bc, cd, abc, bcd$ e $abcd$ são todas as subpalavras da palavra $abcd$. Um **prefixo** de uma palavra é qualquer número de símbolos iniciais da palavra, e um **sufixo** é qualquer número de símbolos finais. Por exemplo, a palavra abc tem prefixos ϵ, a, ab e abc ; seus sufixos são ϵ, c, bc e abc . Uma palavra pode ter várias ocorrências da mesma subpalavra. Por exemplo, a palavra $ababab$ tem três ocorrências da subpalavra ab e duas da subpalavra $abab$. Para cada palavra w e cada número natural i , a palavra w^i é definida indutivamente por: (1) $w^0 = \epsilon$, a palavra vazia e (2) $w^{i+1} = w^i \cdot w$, para cada $i \geq 0$. Assim, por exemplo, $w^1 = w$ e $(bac)^2 = bacbac$.

Uma **linguagem (formal)** é um conjunto de palavras de símbolos de algum alfabeto. Um exemplo de linguagem é o conjunto de todas as palavras sobre um alfabeto Σ . Esta linguagem é denotada por Σ^* . Por exemplo, se $\Sigma = \{a\}$, então $\Sigma^* = \{\epsilon, a, aa, aaa, \dots\}$. Se $\Sigma = \{0, 1\}$, então $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$.

2.2 Computação Paralela

O propósito desta seção é apresentar uma pequena introdução ao modelo paralelo de memória compartilhada juntamente com a especificação de uma estrutura recente, apropriada para descrição e análise de algoritmos paralelos, denominada Estrutura Trabalho-Tempo de Apresentação. Simulações entre as diversas variantes do modelo de memória compartilhada também serão descritas, mostrando que não existe uma diferença significativa entre os poderes relativos dessas variantes. Além disso, apresentaremos a noção de P -completude e suas relações intrínsecas com a classe de complexidade NC de algoritmos paralelos. Estaremos assim apresentando alguns conceitos básicos sobre computação paralela, baseados no livro de JáJá [JáJ92].

Um modelo quase unânime para projeto e análise de algoritmos seqüenciais consiste de uma unidade central de processamento conectada a uma memória de acesso aleatório¹. O conjunto de instruções comum utilizado neste modelo inclui leitura e escrita na memória e operações lógicas e aritméticas básicas. O sucesso desse modelo se deve a sua simplicidade e a sua capacidade de incorporar o desempenho de algoritmos seqüenciais sobre os computadores do tipo von Neumann.

Infelizmente, a computação paralela não possui um modelo algorítmico amplamente aceito. A razão principal para a não existência de tal modelo se deve ao fato de que o desempenho dos algoritmos paralelos depende de um conjunto de fatores intrinsecamente relacionados, dependente da máquina. Estes fatores incluem concorrência computacional, alocação e escalonamento de processadores, comunicação e sincronização.

¹Modelo RAM (*Random Access Machine*)

2.2.1 Processamento Paralelo

O principal objetivo do processamento paralelo é executar computações, com a utilização de processadores trabalhando concorrentemente, de uma forma mais rápida que as computações que utilizam um único processador. A busca constante desse objetivo vem tendo um forte impacto em praticamente todas as atividades relacionadas à ciência da computação. A necessidade de se encontrar soluções cada vez mais rápidas e de se resolver problemas com grandes volumes de informações surge freqüentemente em uma ampla variedade de aplicações tais como previsão de tempo, processamento de imagens, dinâmica dos fluidos, biologia molecular, etc.

Existem dois fatores principais que têm contribuído para a tendência atual em favor do processamento paralelo, levando pesquisadores a explorar o paralelismo e seu uso potencial em aplicações importantes:

- o custo do *hardware* vem caindo ininterruptamente, possibilitando o projeto de sistemas de computação com muitos processadores a um custo razoável;
- desenvolvimento constante da tecnologia de integração de circuitos em altíssima escala (*Very Large Scale Integration* - *VLSI*), fazendo com que se torne possível o projeto de sistemas de computação bastante complexos, envolvendo milhões de transistores em uma única pastilha.

Um **computador paralelo** é uma coleção de processadores, usualmente do mesmo tipo, interconectados de uma certa maneira que permite a coordenação de suas atividades e a troca de informações. Em um computador paralelo, os processadores estão dispostos a uma pequena distância um do outro e são utilizados para solucionar um problema em conjunto.

Os computadores paralelos podem ser classificados de acordo com uma variedade de características arquiteturais e modos de operação. Estes critérios de classificação incluem o tipo e a quantidade de processadores, as interconexões entre os processadores e os esquemas de comunicação correspondentes, o controle e a sincronização e as operações de leitura e escrita.

Apresentaremos em nossa dissertação algoritmos que são apropriados para implementação em computadores paralelos. Existem critérios de avaliação importantes para que um algoritmo seqüencial seja adequadamente implementável tais como desempenho de tempo, utilização de espaço e facilidade de programação. Os critérios de avaliação para um algoritmo paralelo são mais complexos devido à presença de parâmetros adicionais tais como a quantidade de processadores, as capacidades das memórias locais e a comunicação. Medidas de avaliação de desempenho de algoritmos paralelos que utilizaremos em nosso trabalho serão apresentadas na Seção 2.2.4.

2.2.2 Modelos de Computação Paralela

Os modelos de computação paralela são consideravelmente mais complexos que os modelos sequenciais devido, em especial, à presença de muitos processadores interconectados. O modelo de memória compartilhada que destacaremos a seguir é um modelo que pode ser utilizado como uma estrutura geral para descrever e analisar algoritmos paralelos. A descrição do modelo do grafo orientado acíclico (*directed acyclic graph - dag*), do modelo de rede, do modelo paralelo de árvore de comparação, do modelo de redes de ordenação e do modelo de circuitos booleanos são apresentadas em [JáJ92]. Tais descrições não fazem parte de nosso trabalho. Muitos conceitos apresentados a seguir são baseados em [JáJ92].

Um modelo de computação paralela deveria, de uma maneira geral, satisfazer as seguintes propriedades, infelizmente conflitantes:

1. **Simplicidade:** o modelo deve ser simples o suficiente para permitir a descrição de algoritmos paralelos facilmente e para analisar matematicamente medidas de desempenho importantes tais como tempo, comunicação e utilização de espaço. Além disso, o modelo não deve ser dependente de qualquer classe de arquiteturas e portanto, deve ser tão independente do *hardware* quanto possível;
2. **Implementabilidade:** os algoritmos paralelos desenvolvidos para um modelo determinado devem ser facilmente implementáveis em computadores paralelos. Além disso, qualquer análise deve capturar de forma significativa o desempenho real desses algoritmos em computadores paralelos.

Apesar de todo desenvolvimento no sentido de se obter tais propriedades, nenhum modelo paralelo tem provado ser aceito por uma maioria de pesquisadores em processamento paralelo.

Os algoritmos apresentados em nossa dissertação pertencem ao modelo paralelo de memória compartilhada. Nesse modelo, uma extensão natural do modelo sequencial conhecido, muitos processadores têm acesso a uma memória global, chamada memória compartilhada. Ou melhor, o modelo paralelo de memória compartilhada consiste de uma certa quantidade de processadores, cada um contendo sua memória local e podendo executar o seu próprio programa local, de tal forma que todos os processadores se comunicam por troca de dados através de uma memória compartilhada. A memória compartilhada também é chamada de memória global. A Figura 2.1 mostra uma visão geral do modelo de memória compartilhada.

Existem dois modos básicos de operação dos processadores em um modelo de memória compartilhada. No primeiro modo, chamado **síncrono**, todos os processadores operam sincronamente sob o controle de um relógio comum. O nome padrão para o modelo síncrono de memória compartilhada é **modelo de máquina paralela de acesso**

Figura 2.1: O modelo de memória compartilhada. Cada processador p_i é identificado com um índice i chamado de número do processador ou identificação do processador.

aleatório (*parallel random access machine (PRAM) model*). No segundo modo, chamado **assíncrono**, cada processador opera sob um relógio separado, sendo de responsabilidade do programador fixar pontos de sincronização apropriados, sempre que necessários.

Como cada processador pode executar seu próprio programa local, o modelo de memória compartilhada é do tipo **múltiplas instruções e múltiplos dados** (*multiple instructions multiple data (MIMD) type*). Isto é, cada processador pode executar uma instrução distinta daquela executada por qualquer outro processador, em qualquer unidade de tempo. Ou ainda, pode operar sobre dados distintos daqueles operados por qualquer outro processador, durante qualquer unidade de tempo. Para um dado algoritmo, a quantidade de dados transferidos entre a memória compartilhada e as memórias locais dos diferentes processadores representa a quantidade de **comunicação** requerida pelo algoritmo. Em nossa dissertação, os algoritmos paralelos que apresentaremos estão descritos no modelo síncrono de memória compartilhada ou no modelo PRAM. Em geral, os algoritmos desenvolvidos para o modelo PRAM são do tipo **única instrução e múltiplos dados** (*single instruction multiple data (SIMD) type*). Ou seja, todos os processadores executam o mesmo programa de maneira que, durante cada unidade de tempo, todos os processadores ativos estão executando a mesma instrução, mas possivelmente operando sobre dados distintos.

Existem muitas variações do modelo PRAM baseadas no acesso dos vários processadores à mesma posição da memória global. A PRAM com **leitura exclusiva e escrita exclusiva** (*exclusive read exclusive write PRAM* ou *EREW PRAM*) não permite qualquer acesso simultâneo a uma localização de memória. A PRAM com **leitura concorrente e escrita exclusiva** (*concurrent read exclusive write PRAM* ou *CREW PRAM*) permite acesso simultâneo somente para a instrução de leitura. O acesso simultâneo a uma posição de memória tanto para uma instrução de leitura como para uma instrução de escrita é permitido no modelo PRAM com **leitura concorrente e escrita concorrente** (*concurrent read concurrent write PRAM* ou *CRCW PRAM*). Existem três variações para o modelo

PRAM CRCW de leitura concorrente e escrita concorrente, diferenciadas pela maneira como a escrita concorrente é executada. O modelo PRAM CRCW **comum** permite escrita concorrente somente quando todos os processadores estão tentando escrever o mesmo valor. O modelo PRAM CRCW **arbitrário** permite, quando existe concorrência na instrução de escrita, um processador qualquer realizar a escrita. O modelo PRAM CRCW **prioritário** supõe que os identificadores dos processadores estão linearmente ordenados e permite que o processador de menor identificador escreva.

2.2.3 Estrutura Trabalho-Tempo de Apresentação

O modelo PRAM ajuda-nos a descrever algoritmos paralelos simplificadaamente, evitando inúmeros detalhes contidos com frequência nesses algoritmos. O paradigma de descrição que apresentaremos a seguir nos ajudará ainda mais nesse sentido.

O **paradigma trabalho-tempo (TT)** fornece orientações informais para uma descrição *top-down* de algoritmos paralelos em dois níveis. O nível superior suprime detalhes específicos do algoritmo. O nível inferior segue o Princípio Geral de Escalonamento de Brent, a ser apresentado a seguir, e resulta em uma descrição completa para o modelo PRAM.

O **Nível Superior**, também chamado de **Apresentação Trabalho-Tempo (TT) de Algoritmos Paralelos**, estabelece o seguinte objetivo para o projeto de um algoritmo paralelo: descrever o algoritmo em termos de uma seqüência de unidades de tempo, onde cada unidade de tempo pode incluir qualquer número de operações concorrentes. O **trabalho** $\tau(n)$ executado por um algoritmo paralelo é definido como sendo o número total de operações utilizadas por ele, sobre uma entrada de tamanho n .

Antes de prosseguir com a apresentação desse paradigma, introduziremos um comando muito útil, chamado **faça-em-paralelo**, que possibilita uma descrição simples e clara de algoritmos dentro dessa estrutura:

para $s \leq i \leq t$ **faça-em-paralelo**

seqüência-de-instruções

A *seqüência-de-instruções* depende do índice i . As instruções correspondentes a todos os valores de i entre s e t são executadas em paralelo. Observe que uma seqüência de instruções é caracterizada por sua indentação. A principal vantagem de se descrever algoritmos dessa forma, em relação a uma especificação PRAM, é que não existe a preocupação com os processadores. A presença de p processadores limita o número de operações para no máximo p em cada unidade de tempo. Além disso, existe o problema de alocação desses processadores.

Suponha agora que a apresentação TT de algoritmos fornece um algoritmo paralelo de complexidade de tempo $T(n)$ e de trabalho $\tau(n)$, ou seja, que este algoritmo necessite de um total de $\tau(n)$ operações. O **Nível Inferior** mostra que, utilizando o Princípio Geral de Escalonamento TT de Brent, apresentado a seguir, podemos quase sempre adaptar este algoritmo para ser executado em uma PRAM com p processadores em $\leq \lfloor \frac{\tau(n)}{p} \rfloor + T(n)$ passos paralelos.

O **Princípio de Escalonamento TT** de Brent pode ser descrito da seguinte forma. Seja $\tau_i(n)$ o número de operações executadas na unidade de tempo i , onde $1 \leq i \leq T(n)$. Podemos então simular cada conjunto de $\tau_i(n)$ operações em $\lfloor \frac{\tau_i(n)}{p} \rfloor$ passos paralelos em p processadores, para cada $1 \leq i \leq T(n)$. Se a simulação obtém sucesso, o algoritmo correspondente no modelo PRAM com p processadores utiliza $\leq \sum_i \lceil \frac{\tau_i(n)}{p} \rceil \leq \sum_i (\lfloor \frac{\tau_i(n)}{p} \rfloor + 1) \leq \lfloor \frac{\tau(n)}{p} \rfloor + T(n)$ passos paralelos.

Uma observação importante é que o sucesso deste princípio depende basicamente de duas características de implementação. A primeira é o cálculo de $\tau_i(n)$ para cada i , o que é geralmente imediato. A segunda é a alocação de cada processador para executar um conjunto de tarefas apropriadas.

2.2.4 Otimalidade

Seja $\mathcal{P}$ um problema computacional. O **limitante inferior seqüencial** para o problema $\mathcal{P}$ é definido como sendo uma função $T^*(n)$ tal que a complexidade seqüencial de pior caso de qualquer algoritmo que solucione $\mathcal{P}$ é $\Omega(T^*(n))$, para qualquer entrada de tamanho n . Isto é, todo algoritmo seqüencial que soluciona $\mathcal{P}$ efetua, pelo menos, $\Omega(T^*(n))$ passos seqüenciais. Note que o limitante inferior seqüencial para um problema descreve sua complexidade intrínseca, independentemente dos algoritmos existentes ou que ainda virão a existir. Dessa forma, se existir um algoritmo seqüencial de complexidade de tempo $O(T^*(n))$, esse algoritmo é dito ser de **complexidade de tempo ótimo** ou simplesmente de **tempo ótimo**.

Para os algoritmos paralelos que estaremos descrevendo em nossa dissertação existem dois tipos de otimalidade que apresentaremos em seguida.

Um algoritmo paralelo descrito no nível de apresentação TT, que soluciona $\mathcal{P}$, será chamado de **trabalho ótimo** ou **ótimo no sentido fraco** se o trabalho realizado pelo algoritmo é tal que $\tau(n) = \Theta(T^*(n))$. Em outras palavras, o número total de operações utilizadas pelo algoritmo paralelo ótimo é assintoticamente equivalente à complexidade seqüencial do problema, não importando sua complexidade de tempo paralelo $T(n)$.

Por outro lado, um algoritmo paralelo de trabalho ótimo é de **trabalho-tempo (TT) ótimo** ou **ótimo no sentido forte** se podemos mostrar que sua complexidade de tempo $T(n)$ não pode ser melhorada por qualquer outro algoritmo paralelo de trabalho ótimo.

2.2.5 Simulação entre Modelos PRAM

O modelo de computação utilizado nos algoritmos paralelos que descreveremos no Capítulo 4 é o modelo PRAM CRCW. Nesta seção, baseada também no livro de JáJá [JáJ92], serão exploradas as relações entre os diferentes modelos PRAM. Como resultado, veremos que uma PRAM CRCW prioritária com p processadores, o modelo PRAM mais poderoso, pode ser simulada em uma PRAM EREW com p processadores, o modelo PRAM mais fraco, com um fator de perda de $O(\log p)$. Essa simulação, em geral, não pode ser melhorada. A situação torna-se mais delicada quando se investiga os poderes relativos das diferentes PRAMs CRCW.

É importante observar que nossa intenção nesse ponto é apenas tornar evidente a idéia de que não existem diferenças significativas entre os diversos modelos de memória compartilhada. Como os algoritmos paralelos que estudaremos estão descritos em um dos mais poderosos modelos PRAM, isto é, na PRAM CRCW arbitrária, acreditamos que essa observação seja essencial. No entanto, a apresentação das provas dos resultados de tais simulações não serão apresentadas por nós. Para maiores detalhes, o leitor deve consultar a referência [JáJ92].

Simulação da PRAM CRCW prioritária em uma PRAM EREW

O teorema abaixo, juntamente com o corolário que o segue, caracterizam a simulação de uma PRAM CRCW prioritária com p processadores em uma PRAM EREW com p processadores.

Teorema 2.2.1 *Uma instrução de leitura ou escrita concorrente de uma PRAM CRCW prioritária com p processadores pode ser implementada em uma PRAM EREW com p processadores em tempo $O(\log p)$.*

□

Corolário 2.2.2 *Dado um algoritmo que pode ser implementado em tempo T em uma PRAM CRCW prioritária com p processadores, pode-se implementar esse algoritmo em uma PRAM EREW com p processadores em tempo $O(T \log p)$.*

□

Como qualquer algoritmo descrito em uma PRAM EREW pode ser implementado em uma PRAM CREW sem mudanças significativas, a simulação descrita no Teorema 2.2.1 também implica que a PRAM CREW com p processadores pode simular a PRAM CRCW prioritária com p processadores com complexidade de tempo equivalente.

Pode-se mostrar também que a PRAM EREW com p processadores pode simular uma PRAM CREW com p processadores com as mesmas complexidades apresentadas no Teorema 2.2.1 e no Corolário 2.2.2. E, em geral, essas complexidades que resultam não podem ser melhoradas.

Simulação da PRAM CRCW prioritária em uma PRAM CRCW comum

A relação entre as variantes da PRAM CRCW é mais sutil. Parece claro que qualquer algoritmo projetado para a PRAM CRCW comum será executado corretamente em uma PRAM CRCW arbitrária, e qualquer algoritmo projetado para a PRAM CRCW arbitrária será executado corretamente em uma PRAM CRCW prioritária. Este fato sugere que a PRAM CRCW prioritária é mais poderosa que a PRAM CRCW arbitrária, que por sua vez parece ser mais poderosa que a PRAM CRCW comum. Se, no entanto, não existe qualquer restrição sobre o número de processadores ou sobre a quantidade de memória compartilhada disponível, então os três modelos são equivalentes. Mais precisamente, o resultado a seguir afirma que um passo de escrita concorrente de uma PRAM CRCW prioritária com p processadores pode ser simulado por $O(1)$ passos em uma PRAM CRCW comum com $O(p \log p)$ processadores.

Teorema 2.2.3 *Qualquer algoritmo sendo executado em uma PRAM CRCW prioritária com p processadores em tempo T pode ser simulado em uma PRAM CRCW comum com $O(p \log p)$ processadores em tempo $O(T)$.*

□

Apesar da PRAM CRCW comum poder simular uma PRAM CRCW prioritária em tempo proporcionalmente equivalente, essa simulação não é eficiente já que o número de processadores aumenta por um fator de $\log p$. Além disso, a quantidade de memória compartilhada cresce de, digamos, m localizações na memória global da PRAM CRCW prioritária, para $\Theta(mp)$ na PRAM CRCW comum. É possível classificar e comparar as PRAMs CRCW restringindo-se o número de processadores e a quantidade de memória compartilhada. Isso não será feito aqui.

Assim, finalmente, será apresentada a simulação no caso quando as duas máquinas envolvidas possuem o mesmo número de processadores, mas nenhuma restrição na quantidade de memória compartilhada é feita.

Teorema 2.2.4 *Uma instrução de uma PRAM CRCW prioritária com p processadores pode ser simulada em uma PRAM CRCW comum com p processadores em tempo $O(\log p / \log \log p)$.*

□

Simulação da PRAM CRCW prioritária em uma PRAM CRCW arbitrária

A simulação final que apresentaremos é feita entre a PRAM CRCW prioritária e a PRAM CRCW arbitrária, onde ambas possuem p processadores. Surpreendentemente, cada passo da PRAM CRCW prioritária necessita de no máximo $O(\log \log p)$ passos na PRAM CRCW arbitrária.

Teorema 2.2.5 *Uma instrução de uma PRAM CRCW prioritária com p processadores pode ser simulada em tempo $O(\log \log p)$ em uma PRAM CRCW arbitrária com p processadores.*

□

2.2.6 P -completude

Um dos principais objetivos do processamento paralelo é construir técnicas para projeto de algoritmos paralelos eficientes para uma ampla variedade de problemas. Um objetivo implícito dentro de uma determinada linha de pesquisa é desenvolver algoritmos no modelo PRAM que são muito rápidos, isto é, algoritmos de tempo $O(\log^k n)$ para alguma constante k fixa, e utilizando um número total de operações comparável à complexidade de tempo do melhor algoritmo sequencial conhecido que soluciona tal problema. Entretanto, nem sempre isso é possível. Em geral, os problemas apresentam diferentes graus de “paralelizabilidade”.

A classe de complexidade NC , que será descrita mais formalmente a seguir, consiste de todos os problemas que podem ser solucionados por algoritmos de tempo polilogarítmico em uma PRAM, utilizando um número polinomial de processadores.

Dadas as simulações entre os modelos PRAM descritas nas seções anteriores, a classe NC se mantém invariante sob todos os modelos PRAM. Segue que um problema que não está em NC não pode ser solucionado de forma extremamente rápida com um número polinomial de processadores.

Classe NC

Classes de complexidade são tipicamente definidas em termos de linguagens. Seja L uma linguagem sobre o alfabeto $\Sigma = \{0, 1\}$. Um algoritmo para reconhecer a linguagem L toma uma palavra arbitrária $x \in \Sigma^*$ e determina se x está em L ou não. Dessa forma, o problema de reconhecimento de uma linguagem é um **problema de decisão**, cuja resposta é “sim” ou “não”.

É preferível tratar os problemas de reconhecimento de linguagens, já que a teoria se torna mais simples com a omissão de detalhes supérfluos. Problemas de decisão são facilmente associados a qualquer problema computacional. Em geral, um problema computacional é pelo menos tão difícil de se resolver como qualquer problema de decisão associado a ele. Portanto, evidências indicando que um problema de decisão não é paralelizável podem ser utilizadas para mostrar que o próprio problema computacional não é paralelizável.

A classe NC é definida como sendo o conjunto de todas as linguagens L tais que, sobre todas as entradas de tamanho n , L pode ser reconhecida por um algoritmo PRAM em tempo $O(\log^k n)$, utilizando um número polinomial de processadores, onde k é uma constante que não depende de n .

Classes P e NC

Seja P a classe de todas as linguagens que podem ser reconhecidas por uma máquina de Turing determinística com um número polinomial de passos. A classe P mantém-se invariante sob vários modelos de computação seqüencial, e, em particular, P pode ser definida como a classe de todos os problemas que podem ser resolvidos em uma RAM em tempo polinomial. A classe P é universalmente aceita para representar a classe de todos os problemas que podem ser resolvidos eficientemente em uma máquina de processamento seqüencial.

Existe uma relação intrigante entre as classes P e NC . Por um lado, é fácil ver que $NC \subseteq P$ convertendo os algoritmos em NC em algoritmos seqüenciais de maneira óbvia. Um problema fundamental em teoria das complexidades é se $P \subseteq NC$ ou não. Pelo que parece, temos que $P \not\subseteq NC$, e assim, existem problemas em P que não podem ser solucionados rapidamente em paralelo com um número polinomial de processadores. A classe de problemas P -completos, que será brevemente introduzida, consiste dos mais prováveis candidatos de P que não estão em NC .

Noção de Redutibilidade

Sejam L_1 e L_2 duas linguagens. A linguagem L_1 é NC -redutível à linguagem L_2 se existe um algoritmo em NC que transforma uma entrada arbitrária x_1 de L_1 em uma entrada x_2 de L_2 tal que $x_1 \in L_1$ se e somente se $x_2 \in L_2$.

Note que a noção de redutibilidade não é simétrica e que uma instância arbitrária de L_1 é transformada em uma instância potencialmente especial de L_2 . A existência da transformação NC implica na existência de um algoritmo para reconhecimento de L_1 , dado qualquer algoritmo para reconhecimento de L_2 . Em particular, o seguinte resultado pode ser enunciado.

Lema 2.2.6 *Sejam L_1 e L_2 duas linguagens tais que L_1 é NC -reduzível a L_2 . Se $L_2 \in NC$ então $L_1 \in NC$.*

□

Um importante fato sobre NC -reduzibilidade é dado no lema abaixo, mostrando que essa relação possui a propriedade da transitividade.

Lema 2.2.7 *Sejam L_1, L_2 e L_3 linguagens. Se L_1 é NC -reduzível a L_2 e L_2 é NC -reduzível a L_3 , então L_1 é NC -reduzível a L_3 . Isto é, NC -reduzibilidade é transitiva.*

□

Noção de P -completude

A noção central de P -completude é dada pela seguinte definição. Uma linguagem L é **P -completa** se (1) $L \in P$, e (2) toda linguagem em P é NC -reduzível a L . Dessa definição, o seguinte fato pode ser deduzido.

Lema 2.2.8 *Seja L um problema P -completo. Se $L \in NC$ então $NC = P$.*

□

O Lema 2.2.8 pode ser reescrito da seguinte forma. Se $P \neq NC$, como amplamente conjecturado, então nenhum problema P -completo pertence a NC . Portanto, um problema P -completo representa um provável candidato de P que pode não ser solucionado muito rapidamente em uma PRAM com número polinomial de processadores.

Capítulo 3

Árvore dos Sufixos

Uma grande parte dos problemas de processamento de palavras pode ser descrita dentro do seguinte contexto. Dada uma palavra x , pré-processamos x de modo a estabelecer estruturas de dados eficientes para permitir o processamento rápido de consultas a subpalavras de x . Exemplos de consultas são: encontrar a maior subpalavra repetida de x , encontrar uma ocorrência de uma palavra y em x (casamento de padrões), encontrar, dadas duas subpalavras de x , o maior prefixo compartilhado por elas, etc.

A árvore dos sufixos de [Wei73] é uma estrutura de dados que suporta o processamento dessas consultas, além de muitas outras. Ela foi inicialmente proposta para resolver o problema do casamento de padrões. Existe um conjunto amplo e bem diversificado de aplicações da árvore dos sufixos em problemas de processamento de palavras, compressão de dados, processamento de imagens e, mais recentemente, em biologia computacional.

A árvore dos sufixos está inserida dentro do contexto de busca digital. Dadas n chaves e um valor k correspondente a uma chave, ao contrário da busca por comparações que tenta comparar o valor absoluto da chave k com as n chaves, a busca digital [Knu73] procura representar essas chaves como uma seqüência de símbolos e tenta extrair as informações necessárias dessa representação na busca da chave k .

Outra estrutura pertencente a essa classe de estruturas de busca digital é a árvore *trie* de [Fre60] associada a um conjunto finito de palavras sobre um alfabeto qualquer. Quando uma *trie* está associada ao conjunto dos sufixos de uma palavra x , ela recebe o nome de árvore dos sufixos esticada, que será a base da definição do principal objeto de nosso estudo, a árvore dos sufixos.

Além de um conjunto de definições e propriedades básicas relacionadas a essas árvores, apresentaremos ainda considerações importantes, sob o ponto de vista das suas aplicações, sobre a altura e o tamanho da árvore dos sufixos.

3.1 Definição Inicial

Apresentaremos inicialmente o conceito geral de árvore dos sufixos extraído dos trabalhos que propõem algoritmos paralelos para sua construção. Ou seja, as definições de árvore dos sufixos que encontramos especificamente nesses trabalhos são definições muito semelhantes. Isso nos permite obter uma noção inicial e uniforme do que vem a ser essa estrutura. Mais tarde, apresentaremos uma definição mais formal.

Definição 3.1.1 *Seja x uma palavra de tamanho n , constituída de símbolos $x_1, \dots, x_n$ tal que o símbolo $x_n = \#$ é um símbolo especial que não ocorre em nenhuma outra posição de x . A **árvore dos sufixos** T_x associada à palavra x é uma árvore com raiz, com n folhas e no máximo $n - 1$ vértices internos tal que:*

1. *as arestas de T_x são orientadas no sentido raiz $\rightarrow$ folhas e são rotuladas com subpalavras de x ;*
2. *quaisquer duas arestas adjacentes, isto é, que saem do mesmo vértice, estão rotuladas com subpalavras de prefixos diferentes;*
3. *cada folha está rotulada com um inteiro i , onde $1 \leq i \leq n$, que representa uma posição em x ;*
4. *a concatenação dos rótulos das arestas de um caminho da raiz até uma folha i corresponde ao sufixo de x que começa na posição i .*

A Figura 3.1 ilustra uma árvore dos sufixos T_x associada à palavra $x = abbaca\#$.

Figura 3.1: A árvore dos sufixos T_x associada à palavra $x = abbaca\#$, sobre o alfabeto $\Sigma = \{a, b, c, \#\}$.

A formalização dessa idéia, apresentada na próxima seção, tem por objetivo principal reunir um conjunto de propriedades relativas à árvore dos sufixos, como também construir uma notação específica para que as demonstrações de tais propriedades sejam obtidas de forma clara e precisa.

3.2 Formalização

Apresentaremos agora uma coleção de definições e propriedades formalizadas relativas à árvore dos sufixos, bem como às árvores que lhe dão origem. Grande parte desse conteúdo encontra-se em [Wei73] e em [Knu73]. Por todo tratamento formal que preferimos dar ao assunto, especialmente neste capítulo, optamos por apresentar um conjunto de propriedades, juntamente com suas demonstrações completadas por nós. Essas demonstrações, muito provavelmente pela simplicidade, não aparecem na literatura.

Definição 3.2.1 Uma árvore t -ária $\mathcal{T}$ associada ao alfabeto $\Sigma = \{\sigma_1, \dots, \sigma_t\}$ é um conjunto de vértices $V_{\mathcal{T}}$ que ou é vazio ou consiste de uma raiz $v_0 \in V_{\mathcal{T}}$ e t árvores t -árias ordenadas e disjuntas.

A Figura 3.2 mostra um exemplo de uma árvore t -ária para $t = 3$.

Figura 3.2: A árvore ternária $\mathcal{T} = (V, v_0, \mathcal{S})$ associada ao alfabeto $\Sigma = \{a, b, c\}$. O vértice v é o sucessor $\mathcal{S}(v_0, c)$ de v_0 . O pai $\mathcal{P}(v)$ de v é o vértice raiz v_0 . O grau de saída $d(v)$ de v é 1. O passeio $\mathcal{W}(u)$ de v_0 até u é a palavra aa . Assim, $\tilde{\mathcal{S}}(v_0, aa) = u$.

Definição 3.2.2 *Todo vértice $v_i \in V_{\mathcal{T}} \setminus \{v_0\}$ em $\mathcal{T}$ é raiz de alguma árvore t -ária $\mathcal{T}^i$ que consiste de v_i e mais t árvores t -árias ordenadas e disjuntas, digamos $\mathcal{T}_1^i, \mathcal{T}_2^i, \dots, \mathcal{T}_t^i$. A árvore $\mathcal{T}_j^i$ é chamada uma **sub-árvore** de $\mathcal{T}^i$.*

Definição 3.2.3 *A função sucessor associada a uma árvore t -ária $\mathcal{T}$ é uma função $\mathcal{S}_{\mathcal{T}}$ definida por*

$$\begin{aligned} \mathcal{S}_{\mathcal{T}} : V_{\mathcal{T}} \times \Sigma &\longrightarrow (V_{\mathcal{T}} \setminus \{v_0\}) \cup \{\lambda\} \\ (v_i, \sigma_j) &\longmapsto \begin{cases} v_k, & \text{se } v_k \text{ é a raiz da sub-árvore } \mathcal{T}_j^i = \mathcal{T}^k \\ \lambda, & \text{se } \mathcal{T}_j^i \text{ é vazia} \end{cases} \end{aligned}$$

para todo $v_i \in V_{\mathcal{T}}$ e $\sigma_j \in \Sigma$.

Uma função sucessor $\mathcal{S}_{\mathcal{T}}$ determina completamente uma árvore t -ária $\mathcal{T}$ e escreveremos $\mathcal{T} = (V_{\mathcal{T}}, v_0, \mathcal{S}_{\mathcal{T}})$. Para que a notação apresentada aqui não se torne demasiadamente carregada, evitaremos utilizar o símbolo $\mathcal{T}$ como índice de V , de $\mathcal{S}$ e das funções que serão definidas a seguir, a não ser que esta notação seja realmente necessária para diferenciar funções associadas a árvores distintas.

Definição 3.2.4 *Se $v' = \mathcal{S}(v, \sigma)$ então existe uma **aresta** de v a v' em $\mathcal{T}$ que está rotulada com o símbolo σ . O vértice v é chamado de **pai** de v' e v' é um **filho** de v .*

Definição 3.2.5 *O grau de saída, ou simplesmente **grau**, de um vértice $v \in V$ de uma árvore t -ária $\mathcal{T}$ é uma função d definida por*

$$\begin{aligned} d : V &\longrightarrow \mathbb{N} \\ v &\longmapsto |\{\sigma : \mathcal{S}(v, \sigma) \neq \lambda\}|. \end{aligned}$$

Em outras palavras, o grau d de um vértice $v \in V$ é o número de vértices filhos de v . Assim, o grau $d(v)$ não é necessariamente igual a t , para todo $v \in V$, mas sim, $d(v) \leq t$.

Definição 3.2.6 *Um vértice $v \in V$ tal que $d(v) = 0$ é chamado de **folha** de $\mathcal{T}$. Caso contrário, v é um **vértice interno** de $\mathcal{T}$.*

Definição 3.2.7 *A função sucessor estendida associada a uma árvore t -ária $\mathcal{T}$ é uma função $\tilde{\mathcal{S}} : (V \cup \{\lambda\}) \times \Sigma^* \longrightarrow V \cup \{\lambda\}$ definida indutivamente por*

- i) $\tilde{\mathcal{S}}(\lambda, w) = \lambda$, para toda $w \in \Sigma^*$
- ii) $\tilde{\mathcal{S}}(v, \epsilon) = v$, para todo $v \in V$
- iii) $\tilde{\mathcal{S}}(v, w\sigma) = \mathcal{S}(\tilde{\mathcal{S}}(v, w), \sigma)$, para todo $v \in V, w \in \Sigma^*$ e $\sigma \in \Sigma$.

Observe que nem toda função $\mathcal{S}$ é a função sucessor de uma árvore t -ária. No entanto, uma condição necessária e suficiente para que $\mathcal{S}$ seja uma função sucessor de alguma árvore t -ária pode ser expressa em termos da função sucessor estendida $\tilde{\mathcal{S}}$. A Proposição 3.2.8 abaixo estabelece essa condição.

Proposição 3.2.8 *Seja V um conjunto de vértices tal que $v_0 \in V$ e seja $\mathcal{S} : V \times \Sigma \rightarrow (V \setminus \{v_0\}) \cup \{\lambda\}$ uma função. A função $\mathcal{S}$ é a função sucessor de uma árvore t -ária $\mathcal{T} = (V, v_0, \mathcal{S})$ se e somente se, para todo $v \in V$, existe uma única palavra $w \in \Sigma^*$ tal que $\tilde{\mathcal{S}}(v_0, w) = v$.*

Demonstração.

Provemos ‘se’.

Suponha que para todo vértice $v \in V$ existe uma única palavra $w \in \Sigma^*$ tal que $\tilde{\mathcal{S}}(v_0, w) = v$. Suponha também que a palavra $w = \sigma_{i_1}\sigma_{i_2}\dots\sigma_{i_n}$ tem tamanho n , onde $\sigma_{i_j} \in \Sigma$ e $1 \leq j \leq n$. Então, por ii) e iii) da Definição 3.2.7, temos que:

$$\tilde{\mathcal{S}}(v_0, w) = \mathcal{S}(\mathcal{S}(\dots\mathcal{S}(\mathcal{S}(\tilde{\mathcal{S}}(v_0, \epsilon), \sigma_{i_1}), \sigma_{i_2}) \dots \sigma_{i_{n-1}}), \sigma_{i_n}) = v$$

Dessa forma, a função $\mathcal{S}$ determina completamente o caminho de v_0 até v . Portanto, como o vértice v é um vértice arbitrário em V , temos que $\mathcal{S}$ é a função sucessor da árvore t -ária $\mathcal{T} = (V, v_0, \mathcal{S})$.

Provemos ‘somente se’.

Seja $\mathcal{S}$ a função sucessor de uma árvore t -ária $\mathcal{T} = (V, v_0, \mathcal{S})$ e suponha que existam duas palavras $w, w' \in \Sigma^*$, de tamanhos n e m , respectivamente, tais que $\tilde{\mathcal{S}}(v_0, w) = v = \tilde{\mathcal{S}}(v_0, w')$. Seja $C = v_0, v_1, \dots, v_n = v$ o caminho de v_0 até v rotulado por w . Analogamente, seja $C' = v_0, v'_1, \dots, v'_m = v$ o caminho de v_0 a v rotulado por w' . É fácil ver que as sub-árvores de $\mathcal{T}$, com raízes v_1 e v'_1 , violam a Definição 3.2.1 de uma árvore t -ária $\mathcal{T}$, já que elas não são disjuntas. Portanto, $w = w' \in \Sigma^*$. □

Quando existe uma árvore t -ária $\mathcal{T}$ e uma função $\mathcal{S}$ tal que $\mathcal{T} = (V, v_0, \mathcal{S})$ dizemos que a função sucessor $\mathcal{S}$ é **legítima**. Uma consequência imediata dessa proposição é estabelecida no corolário abaixo.

Corolário 3.2.9 *Se $\mathcal{S}$ é legítima, ou seja, se $\mathcal{S}$ é a função sucessor de uma árvore t -ária $\mathcal{T} = (V, v_0, \mathcal{S})$, então $\mathcal{T}$ é única.* □

Definição 3.2.10 A função pai $\mathcal{P} : V \longrightarrow V$ associada a uma árvore t-ária $\mathcal{T}$ é definida por $\mathcal{P}(v_0) = v_0$ e, para todo $v' \in V \setminus \{v_0\}$,

$$\mathcal{P}(v') = v \iff \mathcal{S}(v, \sigma) = v', \text{ para algum } \sigma \in \Sigma.$$

Definição 3.2.11 Para todo $v \in V$, uma k -composição de $\mathcal{P}$, com $k \geq 0$ inteiro, é definida recursivamente por

$$\begin{cases} \mathcal{P}^k(v) = v, & \text{se } k = 0 \\ \mathcal{P}^k(v) = \mathcal{P}^{k-1}(\mathcal{P}(v)), & \text{se } k > 0 \end{cases}$$

Além disso, o menor inteiro $\tilde{k}$ tal que $\mathcal{P}^{\tilde{k}}(v) = v_0$ é chamado de **nível** ou **profundidade** de v .

Na Figura 3.2, observe que $\mathcal{P}^2(u) = v_0$. Portanto, o vértice u tem profundidade 2.

Definição 3.2.12 Qualquer $v' = \mathcal{P}^k(v)$, para um inteiro $k > 0$, é chamado **ancestral** de v e v é chamado um **descendente** de v' . O vértice raiz v_0 é ancestral de todo vértice $v \in V \setminus \{v_0\}$ da árvore $\mathcal{T}$.

Definição 3.2.13 A função passeio $\mathcal{W} : V \longrightarrow \Sigma^*$ associada a uma árvore t-ária $\mathcal{T}$ é definida recursivamente por

$$\begin{aligned} i) \quad & \mathcal{W}(v_0) = \epsilon \\ ii) \quad & \mathcal{W}(v) = \mathcal{W}(v')\sigma \iff v = \mathcal{S}(v', \sigma). \end{aligned}$$

Proposição 3.2.14 Para todo vértice $v \in V$ e toda palavra $w \in \Sigma^*$,

$$w = \mathcal{W}(v) \iff v = \tilde{\mathcal{S}}(v_0, w).$$

Demonstração.

Segue imediatamente das Definições 3.2.7 e 3.2.13.

□

Proposição 3.2.15 Para todo vértice $v \in V$,

$$v_0 = \mathcal{P}^{\tilde{k}}(v) \iff \tilde{k} = |\mathcal{W}(v)|.$$

Demonstração.

Segue imediatamente das Definições 3.2.7 e 3.2.11.

□

Definição 3.2.16 *Seja $y_1, y_2, \dots, y_m$ um conjunto de palavras sobre um alfabeto $\Sigma = \{\sigma_1, \dots, \sigma_t\}$ tal que y_i não é prefixo de y_j , para todo $i \neq j$, com $1 \leq i, j \leq m$. A árvore **trie** [Fre60], associada ao conjunto de palavras $y_1, \dots, y_m$, é uma árvore t -ária $T = (V, v_0, \mathcal{S})$, com raiz v_0 e m folhas tal que existe uma função bijetiva $\psi : F \rightarrow [1, \dots, m]$, onde $F \subset V$ é o conjunto dos vértices folha de T , tal que se $j = \psi(v)$, para $v \in F$, então $\mathcal{W}(v) = y_j$.*

Quando houver necessidade de distinção, utilizaremos a notação F_T no lugar de F para denotar o conjunto dos vértices folha de T ou de qualquer outra árvore.

Devemos observar que não será importante, dentro do contexto de nosso estudo, que uma árvore *trie* seja ordenada. Dessa forma, a figura a seguir mostra um exemplo de uma árvore *trie* onde a ordenação não é considerada.

Figura 3.3: A árvore *trie* $T = (V, v_0, \mathcal{S})$ associada ao conjunto de palavras $y_1 = aab$, $y_2 = abca$, $y_3 = cbb$, $y_4 = accba$, $y_5 = ca$, sobre o alfabeto $\Sigma = \{a, b, c\}$.

Na verdade, a árvore *trie* $T = (V, v_0, S)$ associada ao conjunto de palavras $y_1, \dots, y_m$ sobre o alfabeto $\Sigma = \{\sigma_1, \dots, \sigma_t\}$, assim como a árvore dos sufixos esticada e árvore dos sufixos que definiremos a seguir, são árvores *t-árias* onde as arestas tais que $S(v, \sigma_j) = \lambda$, para todo $v \in V$ e $\sigma_j \in \Sigma$, não são representadas. Dessa forma, as funções associadas a uma árvore *t-ária*, bem como seu conjunto de propriedades, são válidas também para a árvore *trie*. Além disso, é fácil ver que, sendo a árvore *trie* T uma árvore *t-ária*, não necessariamente ordenada, pela Proposição 3.2.8 e pelo Corolário 3.2.9, a função sucessor S associada à árvore *trie* T é legítima e portanto, T é única a menos de isomorfismo.

Seja x uma palavra de tamanho $|x| = n$ sobre um alfabeto Σ , onde $x_n = \#$ é um símbolo especial que não ocorre em nenhuma outra posição de x .

Definição 3.2.17 A árvore dos sufixos esticada $T_x = (V, v_0, S)$ associada a uma palavra x é uma árvore *trie* associada ao conjunto dos n sufixos de x .

A Figura 3.4 ilustra uma árvore dos sufixos esticada T_x associada à palavra $x = abbaca\#$ de tamanho 7, sobre o alfabeto $\Sigma = \{a, b, c, \#\}$.

Figura 3.4: A árvore dos sufixos esticada $T_x = (V, v_0, S)$ associada ao conjunto de palavras $suf_1 = abbaca\#$, $suf_2 = bbaca\#$, $suf_3 = baca\#$, $suf_4 = aca\#$, $suf_5 = ca\#$, $suf_6 = a\#$ e $suf_7 = \#$, que é o conjunto de todos os sufixos da palavra $x = abbaca\#$, sobre o alfabeto $\Sigma = \{a, b, c, \#\}$.

Note que agora a função ψ é definida como $\psi : F \rightarrow [1, \dots, n]$ e que se $j = \psi(v)$ para alguma folha $v \in F$ então $\mathcal{W}(v) = \text{su}f_j$ é o sufixo da palavra x que começa na posição j .

A árvore dos sufixos esticada é uma particularização da árvore *trie*. Dessa forma, a mesma observação sobre suas funções e propriedades deve ser feita. Em especial, devemos lembrar que a árvore dos sufixos esticada $\mathcal{T}$ é uma árvore *t-ária*, não necessariamente ordenada, e como na Proposição 3.2.8 e no Corolário 3.2.9, temos que a função sucessor $\mathcal{S}$ associada a $\mathcal{T}_x$ é legítima e ainda que a árvore dos sufixos esticada $\mathcal{T}_x$, associada à palavra x , é única a menos de isomorfismo.

Definição 3.2.18 *Seja $\mathcal{T}_x = (V_{\mathcal{T}_x}, v_0, \mathcal{S}_{\mathcal{T}_x})$ a árvore dos sufixos esticada associada a x . A árvore dos sufixos [Wei73] associada a x é uma tripla $T_x = (V_{T_x}, v_0, \mathcal{S}_{T_x})$ onde*

$$V_{T_x} \supset V_{\mathcal{T}_x} = \{v \in V_{\mathcal{T}_x} : v \in F_{\mathcal{T}_x} \text{ ou } d_{\mathcal{T}_x}(v) \geq 2\}$$

e para todo vértice $v \in V_{T_x}$ e todo símbolo $\sigma \in \Sigma$ tal que $\mathcal{S}_{\mathcal{T}_x}(v, \sigma) \neq \lambda$, seja $w = \sigma w' \in \Sigma^*$ a menor subpalavra tal que $\tilde{\mathcal{S}}_{\mathcal{T}_x}(v, \sigma w') = v' \in V_{T_x}$. Então, a função $\mathcal{S}_{T_x}$ é definida por

$$\begin{aligned} \mathcal{S}_{T_x} : V_{T_x} \times \Sigma^* &\rightarrow (V_{T_x} \setminus \{v_0\}) \cup \{\lambda\} \\ (v, \sigma w') &\mapsto \begin{cases} \lambda, & \text{se } \mathcal{S}_{\mathcal{T}_x}(v, \sigma) = \lambda \\ v', & \text{caso contrário.} \end{cases} \end{aligned}$$

A Figura 3.5 ilustra a árvore dos sufixos T_x associada à palavra $x = \text{abbaca}\#$ de tamanho 7, sobre o alfabeto $\Sigma = \{a, b, c, \#\}$. Note que a árvore dos sufixos T_x foi obtida a partir da árvore dos sufixos esticada $\mathcal{T}_x$ da Figura 3.4.

Figura 3.5: A árvore dos sufixos $T_x = (V, v_0, \mathcal{S})$ associada à palavra $x = \text{abbaca}\#$, sobre o alfabeto $\Sigma = \{a, b, c, \#\}$.

A Definição 3.2.18 da árvore dos sufixos T_x associada a uma palavra x está baseada na árvore dos sufixos esticada $\mathcal{T}_x$ associada a x da seguinte forma. O conjunto dos vértices V_{T_x} de T_x é um subconjunto do conjunto de vértices $V_{\mathcal{T}_x}$ de $\mathcal{T}_x$ tal que V_{T_x} contém os vértices folha $F_{\mathcal{T}_x}$ de $\mathcal{T}_x$ e mais os vértices internos de $\mathcal{T}_x$ que possuem mais que um filho. A função sucessor $\mathcal{S}_{T_x}$ de T_x é definida de forma semelhante à função sucessor estendida $\tilde{\mathcal{S}}_{\mathcal{T}_x}$ de $\mathcal{T}_x$. Isso significa que as arestas de T_x estão de fato rotuladas com subpalavras de x , ao contrário das árvores definidas anteriormente, onde uma aresta está rotulada com um único símbolo de um alfabeto. Note ainda que a função ψ é definida como anteriormente: $\psi : F \rightarrow [1, \dots, n]$ e se $j = \psi(v)$ para alguma folha $v \in F$ então $\mathcal{W}(v) = \text{su}f_j$ é o sufixo da palavra x que começa na posição j . A função pai, a k -composição e a função passeio associadas a uma árvore dos sufixos T_x estão baseadas na Definição 3.2.18, onde encontra-se a definição de $\mathcal{S}$, e serão definidas a seguir de forma semelhante.

Definição 3.2.19 A função pai $\mathcal{P} : V \rightarrow V$ associada a uma árvore dos sufixos T_x é definida por $\mathcal{P}(v_0) = v_0$ e, para todo $v' \in V \setminus \{v_0\}$,

$$\mathcal{P}(v') = v \iff \mathcal{S}(v, w) = v', \text{ para alguma } w \in \Sigma^*.$$

Definição 3.2.20 Para todo $v \in V$, uma k -composição de $\mathcal{P}$, com $k \geq 0$ inteiro, é definida recursivamente por

$$\begin{cases} \mathcal{P}^k(v) = v, & \text{se } k = 0 \\ \mathcal{P}^k(v) = \mathcal{P}^{k-1}(\mathcal{P}(v)), & \text{se } k > 0 \end{cases}$$

Além disso, o menor inteiro $\tilde{k}$, tal que $\mathcal{P}^{\tilde{k}}(v) = v_0$, é chamado de **nível** ou **profundidade** de v .

Definição 3.2.21 A função passeio $\mathcal{W} : V \rightarrow \Sigma^*$ associada a uma árvore dos sufixos T_x é definida recursivamente por

$$\begin{aligned} i) \quad & \mathcal{W}(v_0) = \epsilon \\ ii) \quad & \mathcal{W}(v) = \mathcal{W}(v')w \iff v = \mathcal{S}(v', w). \end{aligned}$$

Não é difícil argumentar, fazendo-se uma analogia desta com as árvores definidas anteriormente, que a função sucessor $\mathcal{S}$ de uma árvore dos sufixos T_x , associada a uma palavra x , é legítima e que T_x é única a menos de isomorfismo, conforme a Proposição 3.2.8 e o Corolário 3.2.9.

Observamos finalmente, que a árvore dos sufixos pode ser obtida da árvore dos sufixos esticada pela compactação de seus caminhos de vértices de grau 1, como observado anteriormente e ilustrado nas Figuras 3.4 e 3.5.

3.3 Altura e Tamanho da Árvore dos Sufixos

Devido à necessidade de se percorrer a árvore dos sufixos em praticamente todas as aplicações que a utilizam, sua altura e seu tamanho são informações bastante relevantes. Veremos agora os conceitos de altura e tamanho de uma árvore dos sufixos. Em seguida, apresentaremos seus limitantes triviais, observando que existem trabalhos específicos desenvolvidos com o objetivo de estudar a altura e o tamanho esperados dessa estrutura e, de uma forma geral, estudar seu comportamento esperado, utilizando modelos probabilísticos.

Apresentaremos agora definições mais abrangentes dos conceitos de altura e de tamanho de uma árvore qualquer.

Definição 3.3.1 A altura $h(v)$ de um vértice $v \in V$ de uma árvore T é definida como sendo o tamanho do maior caminho de v até uma folha $f \in F$. Ou seja, $h(v)$ é o número de arestas do maior caminho de v até uma folha f . A altura $h(v_0) = h(T)$ da árvore T é a altura do vértice raiz $v_0 \in V$.

Definição 3.3.2 O tamanho $|V|$ de uma árvore T é definido como sendo a quantidade de vértices de T .

Novamente, vamos considerar T_x uma árvore dos sufixos associada à palavra x de tamanho n , sobre o alfabeto Σ . Os limitantes triviais da altura $h(T_x)$ de T_x são

$$1 \leq h(T_x) \leq n - 1$$

e estão ilustrados na Figura 3.6. Os limitantes do tamanho $|V|$ de T_x são

$$n + 1 \leq |V| \leq 2n - 1$$

e o limitante inferior é um limitante trivial e está ilustrado na Figura 3.6 (a). O limitante superior do tamanho de T_x aparece em várias representações distintas de árvores dos sufixos associadas a palavras diferentes de tamanho n , por exemplo na representação da Figura 3.6 (b). Dessa forma, provaremos que essa desigualdade é verdadeira para qualquer árvore dos sufixos de tamanho n , na Proposição 3.3.3 a seguir.

Proposição 3.3.3 Seja T uma árvore qualquer, orientada, com raiz v_0 e n folhas, tal que $d(v) \geq 2$ para todo vértice $v \in V \setminus F$. Então,

$$|V| \leq 2n - 1.$$

Demonstração.

Por indução na quantidade de folhas $|F| = n$ de T .

Para $n = 1$, temos $|V| = 1 \leq 2 \cdot 1 - 1$.

Para $n = 2$, temos $|V| = 1 + 2 = 3 \leq 2 \cdot 2 - 1$.

Vamos supor que a propriedade seja verdadeira para uma árvore orientada T , com raiz v_0 , com $< n$ folhas, tal que $d(v) \geq 2$, para todo $v \in V \setminus F$.

Vamos provar que para uma árvore orientada T , com raiz v_0 , com n folhas, tal que $d(v) \geq 2$, para todo $v \in V \setminus F$, temos

$$|V| \leq 2n - 1.$$

Seja $v_0 \in V$ o vértice raiz de T tal que $d(v_0) = t$, e sejam $v_1, \dots, v_t$ os vértices tais que $\mathcal{P}(v_1) = \mathcal{P}(v_2) = \dots = \mathcal{P}(v_t) = v_0$. Seja T^i a sub-árvore de raiz v_i de T , para todo i , com $1 \leq i \leq t$. Removendo o vértice raiz v_0 de T , obtemos t árvores T^i , tais que $|F_{T^i}| = n_i < n$, para todo i , com $1 \leq i \leq t$, e $\sum_{i=1}^t n_i = n$. Por hipótese de indução, temos que $|V_{T^i}| \leq 2n_i - 1$, para todo i , com $1 \leq i \leq t$. Assim,

$$\begin{aligned} \sum_{i=1}^t |V_{T^i}| &\leq \sum_{i=1}^t (2n_i - 1) \\ 1 + \sum_{i=1}^t |V_{T^i}| &\leq 1 + \sum_{i=1}^t (2n_i - 1) \\ |V_T| &\leq 1 + 2 \sum_{i=1}^t n_i - \sum_{i=1}^t 1 \\ &\leq 2n - t + 1 \end{aligned}$$

e, como $t = d(v_0) \geq 2$, temos finalmente que

$$|V_T| \leq 2n - 1.$$

□

Antes de dar continuidade a nossa dissertação, devemos observar que existem definições ligeiramente diferentes de árvore dos sufixos na literatura.

A Definição 3.2.17 da árvore dos sufixos esticada e a Definição 3.2.18 da árvore dos sufixos que adotamos, e que fazem parte das publicações que contêm os algoritmos paralelos que estudaremos, não são as mesmas definições descritas em alguns outros trabalhos da área, em especial no estudo específico da altura, do tamanho e do comportamento geral esperado dessa estrutura, construída sobre uma palavra aleatória. A definição de árvore

Figura 3.6: (a) A árvore dos sufixos $T_x = (V, v_0, \mathcal{S})$ associada à palavra $x = abcd\#$ de tamanho $n = 5$, sobre o alfabeto $\Sigma = \{a, b, c, d, \#\}$, de altura $h(T_x) = 1$ e de tamanho $|V| = n + 1 = 6$. (b) A árvore dos sufixos $T_x = (V, v_0, \mathcal{S})$ associada à palavra $x = aaaa\#$ de tamanho $n = 5$, sobre o alfabeto $\Sigma = \{a, \#\}$, de altura $h(T_x) = n - 1 = 4$.

dos sufixos esticada que adotamos é equivalente à definição de árvore dos sufixos descrita nesses trabalhos. Também, a definição de árvore dos sufixos adotada por nós é idêntica à definição de árvore dos sufixos compactada encontrada nesses trabalhos.

Blumer, Ehrenfeucht e Haussler em [BEH89] estudaram o tamanho da árvore dos sufixos T_x associada a uma palavra aleatória x de tamanho n sobre um alfabeto Σ de tamanho t , obtendo uma fórmula em função de t e de n que fornece o tamanho estimado $|V|$ de T_x . A fórmula obtida é uma função não monotonicamente crescente, que oscila entre os limitantes triviais de $|V|$ que apresentamos anteriormente. Ferramentas matemáticas de análise real e de teoria elementar de probabilidades foram utilizadas para a obtenção desse resultado.

Resultados mais significativos sobre a altura foram obtidos especialmente por Szpankowski em [JS91, DSR92, Szp92, Szp93b] e Mahmoud em [Mah92], mostrando que a altura $h(T_x)$ de uma árvore dos sufixos associada a uma palavra aleatória x de tamanho n é proporcional a $\log n$, utilizando propriedades combinatórias de palavras e teoria elementar de probabilidades. Note que esse resultado foi obtido sobre a árvore dos sufixos, que chamamos de árvore dos sufixos esticada. Isso intuitivamente nos faz acreditar que a altura esperada da árvore dos sufixos compactada, que no nosso caso é a árvore dos sufixos, associada a uma palavra aleatória, seja ainda menor. No entanto, o estudo detalhado desse assunto foge dos objetivos de nossa dissertação. Para mais informações, o leitor deve consultar as referências acima citadas.

Capítulo 4

Construção da Árvore dos Sufixos

A árvore dos sufixos, como já mencionado anteriormente, é uma estrutura de dados resultante do pré-processamento de uma palavra x , que permite a realização de consultas rápidas sobre subpalavras de x .

O casamento de padrões (*string matching* ou *pattern matching*) foi o primeiro problema resolvido, por Weiner [Wei73], com a utilização da árvore dos sufixos. Atualmente, diversas tarefas de processamento de palavras fazem uso dessa estrutura e, em consequência, existem vários algoritmos, seqüenciais e paralelos, para a sua construção. Além disso, essa área continua em constante evolução e novos algoritmos estão sendo desenvolvidos com essa finalidade, tais como o algoritmo seqüencial de Ukkonen [Ukk95] e o algoritmo paralelo de Farach, Hariharan e Muthukrishnan [FHM]. Nesse capítulo, nosso objetivo é apresentar algoritmos para a construção da árvore dos sufixos, tentando descrever com detalhes especialmente os algoritmos paralelos.

Inicialmente, faremos alguns comentários breves sobre os algoritmos seqüenciais desenvolvidos por Weiner [Wei73], por McCreight [McC76], por Kosaraju [Kos94], e outros mais recentes propostos por Andersson e Nilsson [AN95], por Kosaraju e Delcher [KD95] e por Ukkonen [Ukk95]. Alguns desses algoritmos, especialmente o algoritmo de McCreight [McC76], deram origem a algoritmos paralelos.

Em seguida, apresentaremos descrições mais detalhadas dos algoritmos paralelos de Landau, Schieber e Vishkin [LSV87], de Apostolico, Iliopoulos, Landau, Schieber e Vishkin [AIL⁺88], de Sahinalp e Vishkin [SV94] e de Hariharan [Har94], sendo que os dois últimos são evoluções recentes e nos apresentam algumas medidas de desempenho ótimas. Apesar do algoritmo de Farach, Hariharan e Muthukrishnan não ter sido ainda publicado, sabemos que, além de apresentar complexidades de trabalho e espaço ótimas, sua complexidade de tempo será melhor que dos algoritmos anteriores. Sempre que possível, estaremos explicitando semelhanças, diferenças, vantagens e desvantagens através de comparações entre esses algoritmos paralelos.

4.1 Algoritmos Seqüenciais

No início da década de 70, o problema do casamento de padrões surgia como um dos grandes desafios da ciência da computação, tornando-se assim um dos problemas mais estudados até nossos dias, devido as suas importantes características práticas e teóricas.

O problema do casamento de padrões consiste basicamente em encontrar uma ocorrência de uma palavra padrão y de tamanho m , em uma palavra texto x de tamanho n , com $m \leq n$, onde ambas as palavras estão definidas sobre um alfabeto Σ . Os algoritmos de casamento de padrões podem ser divididos basicamente em duas classes: a classe dos algoritmos que pré-processam o padrão e então buscam sua ocorrência no texto e a classe que pré-processa o texto, estabelecendo uma estrutura de dados para encontrar uma ocorrência do padrão nessa estrutura. Existem muitos algoritmos eficientes que pertencem à primeira classe. Os algoritmos de nosso interesse estão incluídos na segunda classe.

Weiner, em 1973, propôs um algoritmo de casamento de padrões que se inclui na segunda classe e que constrói uma estrutura de dados denominada por ele de árvore dos sufixos. A fase de pré-processamento, isto é, a construção da árvore dos sufixos associada ao texto x é realizada em tempo $O(n \log |\Sigma|)$, utilizando espaço $O(n)$. Note que o algoritmo de pré-processamento é dependente do tamanho do alfabeto. Note ainda que, se o tamanho do alfabeto for fixo, a complexidade de tempo do algoritmo torna-se linear. Uma vez construída a árvore dos sufixos, as consultas podem ser realizadas com diferentes padrões para encontrar as ocorrências de cada padrão no texto em tempo $O(m \log |\Sigma|)$ por consulta. Apesar de ter sido concebida especialmente para solucionar o problema do casamento de padrões, a árvore dos sufixos transformou-se em uma ferramenta bastante útil para muitas outras tarefas de processamento de palavras.

Em seguida, McCreight em [McC76] propôs um algoritmo para construção da árvore dos sufixos que possui a mesma complexidade de tempo $O(n \log |\Sigma|)$ do algoritmo de Weiner, mas é mais eficiente no espaço gasto. Esse algoritmo mapeia uma palavra x em um índice auxiliar para x na forma de uma árvore *trie*, cujos caminhos da raiz às folhas são sufixos de x e cujos vértices folha correspondem unicamente a posições de x .

Uma breve descrição desse algoritmo é dada a seguir. Na i -ésima iteração, onde $1 \leq i \leq n$, o algoritmo constrói a árvore T_i , através da inserção do i -ésimo sufixo do texto x na árvore T_{i-1} . O algoritmo inicia com a árvore T_0 que possui somente o vértice raiz v_0 . A árvore T_n é a árvore dos sufixos T_x desejada. Note que, se cada iteração é realizada em tempo constante, então a complexidade de tempo total das n iterações do algoritmo é linear. Essa complexidade só é possível, ou seja, cada inserção é realizada em tempo constante, somente se o tamanho do alfabeto Σ é constante. As ligações sufixas adicionadas por McCreight à estrutura de dados representando a árvore dos sufixos garantem essa complexidade de tempo constante por iteração. A cada vértice interno $v \in V$ tal que a palavra σv é um prefixo de $\mathcal{W}(v)$ é acrescentada essa tal ligação sufixa, que parte

do vértice v e aponta para o vértice $u \in V$ tal que w é um prefixo de $\mathcal{W}(u)$. Finalmente, apesar do espaço gasto nos dois algoritmos acima apresentarem ordem de grandeza linear, a estrutura de dados proposta por McCreight é mais econômica no espaço comparando-a com a estrutura proposta por Weiner.

O algoritmo de McCreight tornou-se então bastante utilizado, como um primeiro estágio de pré-processamento, em outros algoritmos de processamento de palavras. Recentemente, por exemplo, Akutsu em [Aku93] propôs um novo algoritmo seqüencial de casamento de padrões, ligeiramente diferente dos anteriores, que utiliza a árvore dos sufixos construída pelo algoritmo de McCreight como primeiro estágio.

Kosaraju, na versão preliminar de um trabalho [Kos94], propôs um algoritmo de casamento de padrões de tempo real, onde a construção da árvore dos sufixos na primeira etapa do casamento de padrões é realizada em tempo *tempo quase real*. Esse algoritmo foi desenvolvido a partir das idéias do algoritmo de McCreight.

Algoritmos mais recentes foram desenvolvidos por Andersson e Nilsson [AN95], por Kosaraju e Delcher [KD95] e por Ukkonen [Ukk95], sendo que este último possibilita que a construção da árvore dos sufixos seja realizada *on-line*. Além disso, novos algoritmos seqüenciais para a construção dessa estrutura vêm sendo desenvolvidos a todo instante. Segue abaixo uma relação de publicações, a qual ainda não tivemos acesso:

1. Kosaraju, *Faster Algorithms for the Construction of Parameterized Suffix Trees*, In *Proceedings, 36th IEEE Symposium on Foundations of Computer Science, FOCS 95*.
2. Cobbs, *Fast Approximate Matching using Suffix Trees*, In *Proceedings, 22nd Annual International Colloquium on Automata, Languages and Programming, ICALP 95*.
3. Giancarlo and Grossi, *On the Construction of Classes of Suffix Trees for Square Matrices: Algorithms and Applications*, In *Proceedings, 22nd Annual International Colloquium on Automata, Languages and Programming, ICALP 95*.
4. Karkkainen, *Suffix Cactus: A Cross between Suffix Trees and Suffix Array*, In *Proceedings, 22nd Annual International Colloquium on Automata, Languages and Programming, ICALP 95*.

4.2 Algoritmos Paralelos

Nosso principal interesse neste capítulo é apresentar um estudo sobre algoritmos paralelos para construção da árvore dos sufixos. Um trabalho pioneiro nessa área foi desenvolvido por Landau e Vishkin em 1986. O algoritmo proposto tem complexidade de tempo $O(\log n)$, utilizando $O(n^2 / \log n)$ processadores, no modelo PRAM CRCW.

Em 1987, Apostolico e Iliopoulos, e Landau, Schieber e Vishkin desenvolveram simultaneamente algoritmos semelhantes para construção da árvore dos sufixos, que foram publicados na forma de relatórios técnicos em suas respectivas universidades. O algoritmo de Landau, Schieber e Vishkin [LSV87] será brevemente descrito na Seção 4.2.1. Em seguida, um trabalho mais completo e detalhado foi publicado por Apostolico, Iliopoulos, Landau, Schieber e Vishkin em [AIL⁺88], resultado da reunião das principais vantagens dos trabalhos anteriores. Este algoritmo será apresentado detalhadamente na Seção 4.2.2.

Esses algoritmos compartilham uma idéia fundamental: utilizam, como um primeiro estágio na construção da árvore dos sufixos, o procedimento paralelo de atribuição de nomes, baseado em um dos primeiros algoritmos seqüenciais para o casamento de padrões, descrito em [KMR72]. A utilidade desse procedimento no projeto de algoritmos paralelos eficientes para processamento de palavras é discutida em [CR91].

Finalmente, uma pequena motivação para o surgimento de algoritmos mais recentes, como os algoritmos de Sahinalp e Vishkin [SV94] e de Hariharan [Har94] será mostrada antes da descrição genérica de cada um deles. Esses novos algoritmos têm como principal característica a otimalidade no sentido fraco, ou seja, o trabalho realizado por eles é equivalente à complexidade de tempo do melhor algoritmo seqüencial conhecido. Ainda, o espaço total utilizado também é ótimo em [Har94], como veremos.

4.2.1 Algoritmo Landau-Schieber-Vishkin

Landau, Schieber e Vishkin em [LSV87] propuseram um algoritmo paralelo para a construção da árvore dos sufixos T_x , associada a uma palavra x de tamanho n , com complexidade de tempo $O(\log n)$, utilizando um total de $O(n \log n)$ operações. Esse algoritmo é descrito em dois estágios, supondo o símbolo $x_n = \#$ um símbolo especial que não ocorre em nenhuma outra posição de x , supondo n uma potência de 2, e supondo também Σ um alfabeto de tamanho fixo.

O primeiro estágio consiste da atribuição de nomes às subpalavras de x de tamanho 2^i , $0 \leq i \leq \log n$, de tal forma que subpalavras iguais sejam identificadas com o mesmo nome. O conjunto de todas as subpalavras de tamanho potência de 2 de uma palavra x é chamado de **conjunto dos fatores básicos de x** . A idéia do procedimento paralelo de atribuição de nomes está contida no algoritmo seqüencial de casamento padrões de Karp, Miller e Rosenberg em [KMR72]. A complexidade de tempo paralelo desse primeiro estágio é $O(\log n)$ e o trabalho realizado é $O(n \log n)$. Pela necessidade do uso de uma matriz $n \times n + 1$ auxiliar nesse procedimento, onde são realizadas leituras e escritas possivelmente concorrentes, e também pela forma como a árvore dos sufixos é armazenada, o espaço total utilizado é $O(n^2)$.

No segundo estágio o **algoritmo Landau-Schieber-Vishkin** constrói propriamente a árvore dos sufixos. Essencialmente, a construção baseia-se na seguinte motivação. A

Figura 4.1: Um pequeno exemplo de execução do **algoritmo Landau-Schieber-Vishkin** para a construção da árvore dos sufixos T_x associada à palavra $x = aab\#$ de tamanho $n = 4$. (a) A árvore T_0 . (b) A árvore T_1 obtida na iteração 1, através do refinamento da árvore T_0 . Note que em T_1 não existem arestas adjacentes rotuladas com palavras de prefixos de tamanho $n/2^1$ iguais. (c) A árvore $T_2 = T_{\log n} = T_x$ obtida na iteração 2, através do refinamento da árvore T_1 . Note também que em T_2 não existem arestas adjacentes rotuladas com palavras de prefixos de tamanho $n/2^2$ iguais. Essa condição final faz com que $T_2 = T_{\log n}$ satisfaça a definição de T_x .

noção de árvore dos sufixos captura tanto as similaridades quanto as diferenças dentro do conjunto dos sufixos de x . Por um lado, o fato de cada sufixo corresponder a um caminho diferente da raiz da árvore até uma folha evidencia as diferenças entre os sufixos. Por outro lado, o fato de dois caminhos da raiz até duas folhas se separarem somente depois que o maior prefixo comum foi compartilhado atesta a similaridade dos sufixos correspondentes. Assim, o segundo estágio começa com uma árvore T_0 que captura somente as diferenças entre os sufixos, negligenciando completamente as suas similaridades. Esse estágio tem $\log n$ iterações tais que, na i -ésima iteração ocorre a construção da árvore T_i , que refina as similaridades capturadas na árvore T_{i-1} . A árvore T_i deve satisfazer a propriedade de que quaisquer duas arestas adjacentes em T_i devem estar rotuladas com subpalavras de x de prefixos de tamanho $n/2^i$ diferentes. A árvore $T_{\log n}$ é a árvore dos sufixos T_x associada à x . Cada refinamento $T_{i-1} \rightarrow T_i$, com $1 \leq i \leq \log n$, pode ser produzido em tempo constante, e portanto a complexidade de tempo do estágio 2 é $O(\log n)$. O trabalho total realizado nesse estágio é $O(n \log n)$. A Figura 4.1 ilustra um esboço de execução do **algoritmo Landau-Schieber-Vishkin**, omitindo seu primeiro estágio e muitos outros detalhes de implementação.

Dessa forma, o tempo total gasto pelo algoritmo paralelo proposto é $O(\log n)$, o trabalho total realizado é $O(n \log n)$ e o espaço total gasto é $O(n^2)$. O espaço total utilizado ainda pode ser reduzido, utilizando uma sugestão dos autores, para $O(n^{1+\delta})$, para qualquer $0 < \delta \leq 1$, à custa do incremento de um fator de $1/\delta$ na complexidade de tempo paralelo do algoritmo. Essa idéia será revista e descrita na próxima seção. A descrição desse algoritmo, conforme [LSV87], no modelo PRAM CRCW arbitrário, utiliza exatos $2n$ processadores. Um esquema para alocação e escalonamento dos processadores também é brevemente descrito.

Acreditamos que o **algoritmo Landau-Schieber-Vishkin** é bastante simples e claro. No entanto, detalhes importantes de implementação, contidos no **algoritmo AILSV** que veremos a seguir na Seção 4.2.2, são omitidos em sua descrição. Mais que isso, existem pequenos melhoramentos obtidos nesse último algoritmo e, por esses motivos, preferimos apresentar apenas esta breve descrição do **algoritmo Landau-Schieber-Vishkin**. Uma descrição completa e bem detalhada do **algoritmo AILSV** será apresentada na Seção 4.2.2 a seguir.

4.2.2 Algoritmo AILSV

Como já mencionamos, trabalhos simultâneos de Apostolico e Iliopoulos e de Landau, Schieber e Vishkin foram apresentados em 1987, descrevendo algoritmos paralelos semelhantes para a construção da árvore dos sufixos T_x associada a uma palavra x de tamanho n , sobre um alfabeto de tamanho constante, com complexidades de tempo, trabalho e espaço equivalentes.

Em seguida, em 1988, Apostolico, Iliopoulos, Landau, Schieber e Vishkin [AIL⁺88] apresentaram um trabalho mais completo, propondo um algoritmo mais detalhado para construção da árvore dos sufixos que reúne as vantagens de cada um dos algoritmos anteriores. Apesar disso, as complexidades desse algoritmo posterior continuam equivalentes às complexidades dos algoritmos anteriores.

O algoritmo de Apostolico et. al [AIL⁺88] constrói a árvore dos sufixos T_x associada a uma palavra x de tamanho n , sobre um alfabeto Σ de tamanho constante, em tempo paralelo $O(\log n)$, realizando um total de $O(n \log n)$ operações, sendo $O(n^2)$ o espaço total utilizado. O algoritmo é um conjunto de procedimentos paralelos que nós apresentaremos na Estrutura Trabalho-Tempo de Apresentação de Algoritmos Paralelos. Observações sobre sua descrição no modelo PRAM CRCW arbitrário, conforme [AIL⁺88], serão devidamente descritas no final desta seção.

Seja $x = x_1x_2 \dots x_n$ uma palavra de tamanho n sobre um alfabeto Σ de tamanho constante. Supomos aqui que $x_n = \#$ é um símbolo especial que não ocorre em nenhuma outra posição de x .

A técnica de construção da árvore dos sufixos T_x contida nesse trabalho consiste também de dois estágios principais. No primeiro estágio, ocorre a construção de uma versão aproximada da árvore dos sufixos, chamada *árvore esqueleto*, em conjunto com o procedimento paralelo de atribuição de nomes aos fatores básicos de x . O segundo estágio consiste do refinamento da árvore esqueleto, ocorrendo então a sua transformação na árvore dos sufixos.

Construção da Árvore Esqueleto D_x

Inicialmente, devemos supor, sem perda de generalidade, que $|x| = n = 2^k$, para algum inteiro positivo k . Além disso, por motivos técnicos observados durante a execução do procedimento que será descrito, estenderemos x adicionando $n - 1$ símbolos especiais $\#$ ao seu final. Essa nova palavra, de tamanho $2n - 1$, é chamada de $x\#$.

A árvore esqueleto $D_x = (V, v_0, \mathcal{S})$ é uma árvore orientada, com raiz v_0 , tal que sua função sucessor $\mathcal{S}$, a função pai $\mathcal{P}$, a k -composição $\mathcal{P}^k$ e a função passeio $\mathcal{W}$ são definidas equivalentemente às funções associadas à árvore dos sufixos T_x . Nesse sentido, a árvore esqueleto D_x é bastante semelhante à árvore dos sufixos T_x , a menos de duas propriedades relevantes: a orientação das arestas e a atribuição de rótulos de prefixos diferentes às arestas adjacentes. Apresentamos a seguir uma definição inicial necessária, de descritores de subpalavras. Além disso, descrevemos um conjunto de características importantes da árvore esqueleto D_x e, em seguida, apresentamos a definição construtiva de [AIL⁺88] para a árvore esqueleto D_x .

Definição 4.2.1 *Seja y uma subpalavra da palavra x . Um **descritor** de y é um par de inteiros (i, l) tal que a subpalavra de tamanho l que começa na posição i de x é idêntica a y .*

A árvore esqueleto D_x é uma árvore com arestas orientadas no sentido *folhas* $\rightarrow$ *raiz*. Apesar disso, já que a construção da árvore esqueleto é *top-down*, então para toda aresta $v \rightarrow u$ em D_x , com $u, v \in V$, consideraremos que $u = \mathcal{P}(v)$, e não o contrário. Note que, pela maneira como desenhamos qualquer árvore, essa consideração parece bastante razoável. Observe também que essa consideração influencia todas as outras funções associadas a D_x . Para todo $v \in V$, temos que $d(v) \geq 2$ e para todo vértice $v \in V \setminus \{v_0\}$ temos que o rótulo da aresta $v \rightarrow \mathcal{P}(v)$ está armazenado no vértice v . Este rótulo é um descritor (i, l) de alguma subpalavra w de $x\#$, com $1 \leq i \leq n$ e $l = 2^q$, para algum $0 \leq q \leq \log n$, tal que $\mathcal{W}(v) = w$. O valor q é o nível de v em D_x .

Um exemplo de uma árvore esqueleto D_x associada à palavra $x = \text{abbabbabbabbaab}\#$ de tamanho $|x| = n = 16$, sobre o alfabeto $\Sigma = \{a, b, \#\}$ está ilustrado na Figura 4.2. Note que foram adicionados $n - 1$ símbolos especiais $\#$ ao final da palavra x , ou seja, o exemplo ilustra a árvore esqueleto D_x associada à palavra $x\#$.

Figura 4.2: A árvore esqueleto D_x associada à palavra $x = abbabbabbabbaab\#$. Observe que $u = \mathcal{P}(v)$. O passeio da raiz v_0 até o vértice v é a palavra $\mathcal{W}(v) = w = abba$ e o tamanho da palavra w é $|w| = 2^q$, onde $q = 2$ é o nível de v em D_x . Observe ainda que a quantidade de símbolos especiais $\#$ em um sufixo suf_i de x é exatamente igual a i .

Definição 4.2.2 A árvore esqueleto $D_x = (V, v_0, \mathcal{S})$ é definida construtivamente por [AIL⁺88] da seguinte forma:

1. O passeio até a raiz v_0 é a palavra vazia ϵ . Isto é, $\mathcal{W}(v_0) = \epsilon$. Além disso, v_0 tem $|\Sigma|$ filhos tais que o passeio até um vértice filho de v_0 corresponde a um símbolo distinto de Σ . Cada vértice $v_i \in V$ tal que $\mathcal{P}(v_i) = v_0$ é rotulado com um descritor $(j, 1)$ tal que $\mathcal{S}(v_0, \sigma_j) = v_i$, com $\sigma_j \in \Sigma$, sendo que quaisquer dois vértices $v_i, v'_i \in V$ distintos, com $\mathcal{P}(v_i) = \mathcal{P}(v'_i) = v_0$, estão rotulados com descritores diferentes.
2. Suponha que todos os vértices do nível $l - 1 \geq 0$ foram inseridos em D_x . Para expandir D_x para o nível $l \leq \log n$, considere todos os vértices do nível $l - 1$. Para um vértice v qualquer deste nível, seja $w = \mathcal{W}(v)$. Então:
 - (a) se $w = z\#$, para alguma palavra z sobre Σ , então faça v a única folha rotulada com o descritor (i, n) , onde i é a primeira componente do rótulo anterior de v ;
 - (b) se $w \neq z\#$, para qualquer palavra z sobre Σ , então seja $S_w = \{s_i : |s_i| = 2 \cdot |w| \text{ e } w \text{ é prefixo de } s_i\}$ o conjunto de todas as palavras com o dobro do tamanho de w e das quais w é prefixo. Pode-se então distinguir dois casos:

- i. se $|S_w| = k > 1$ então crie k filhos $v_1, v_2, \dots, v_k$ de v e faça $\mathcal{W}(v_i) = s_i$, para todo $i, 1 \leq i \leq k$. Ou seja, cada vértice v_i é rotulado com o descritor da subpalavra $s_i = \mathcal{W}(v_i)$;
- ii. se $|S_w| = 1$ então isso significa que w sempre ocorre como prefixo da mesma subpalavra s_1 . Assim, faça $\mathcal{W}(v) = s_1$, isto é, rotule v com o descritor de $s_1 = \mathcal{W}(v)$.

Uma construção paralela natural de D_x é baseada na Definição 4.2.2. Descreveremos a seguir um algoritmo, na Estrutura Trabalho-Tempo de Apresentação de Algoritmos Paralelos, para construção da árvore esqueleto D_x . Esse procedimento efetua o preenchimento de duas estruturas principais: a matriz DS dos descritores das subpalavras de tamanho potência de 2 de $x\#$ e a matriz $Vért$ dos vértices $v \in V$ de D_x .

A matriz DS , de dimensão $n \times (\log n + 1)$, será preenchida com os descritores das subpalavras de $x\#$ de tamanho potência de 2. Para $1 \leq i \leq n$ e $0 \leq q \leq \log n$, $DS[i, q]$ conterá a primeira componente do descritor de uma subpalavra de $x\#$ que inicia na posição i de $x\#$ e tem tamanho 2^q . Por conveniência, supomos $DS[i, q] = n + 1$, para $i > n$.

Cada posição da matriz $Vért$, de dimensão $n \times (\log n + 1)$, é composta por uma estrutura com dois atributos: *Rótulo* e *Pai*. Assim, se para algum valor de $q < \log n$, temos que $Vért[i, q]$ não está vazio, então essa posição representa um vértice $v \in V$ de nível q em D_x , da seguinte forma: o atributo $Vért[i, q].Rótulo$ é uma cópia de $DS[i, q]$ e o atributo $Vért[i, q].Pai$ aponta para a localização de $\mathcal{P}(v)$ em D_x , isto é, $Vért[i, q].Pai$ contém um par de coordenadas que indicam a localização do vértice pai de $Vért[i, q]$ na matriz $Vért$. Além disso, para todo i , onde $1 \leq i \leq n$, $Vért[i, \log n]$ armazena uma folha $f \in F$ de D_x rotulada com o descritor (i, n) . Ou seja, $Vért[i, \log n]$ está preenchido com algum valor, para todo i , onde $1 \leq i \leq n$. Como a Definição 4.2.2 de D_x satisfaz as hipóteses da Proposição 3.3.3, então D_x possui no máximo $2n - 1$ vértices. Assim, podemos representar os vértices de D_x na estrutura $Vért$, de tamanho $n(\log n + 1)$, para todo $n \geq 1$.

O procedimento faz uso ainda de duas estruturas auxiliares: a matriz BB (*bulletin board*), de dimensão $n \times (n + 1)$, auxiliar na atribuição de nomes e onde são realizadas leituras e escritas possivelmente concorrentes, e o vetor Adj , de dimensão n , auxiliar na verificação de arestas adjacentes.

O procedimento *Árvore Esqueleto* recebe como entrada a palavra $x\#$ e uma localização da memória global chamada v_0 , e computa concorrentemente $DS[i, q]$ e $Vért[i, q]$, para todo i , onde $1 \leq i \leq n$ e todo q , onde $1 \leq q \leq \log n$. Primeiramente, algumas inicializações são realizadas, implementando a primeira parte da Definição 4.2.2 de D_x e, em seguida, na principal parte do procedimento, $\log n$ iterações são executadas para implementar a segunda parte dessa definição. Incluímos também neste procedimento a idéia de verificação de arestas adjacentes, somente esboçada em [AIL⁺88]. Finalmente, observamos que o alfabeto Σ deve ser um conjunto de símbolos linearmente ordenado, e que $ord_\Sigma(x_i)$ indica a ordem do símbolo x_i no alfabeto Σ , para todo símbolo x_i de x .

Procedimento *Árvore Esqueleto*

Entrada: a palavra $x\#$ e a raiz v_0 .

Saída: a matriz DS preenchida com os descritores dos fatores básicos de $x\#$ e a matriz $Vért$ preenchida com informações sobre o *Rótulo* e o *Pai* de cada vértice da árvore esqueleto D_x .

01. **Início**
02. **para** $q = 0$ até $\log n$ **faça**
03. **para** $1 \leq i \leq n$ **faça-em-paralelo**
04. $Vért[i, q] \leftarrow \lambda$
05. **para** $1 \leq i \leq n$ **faça-em-paralelo**
06. $BB[1, ord_\Sigma(x_i)] \leftarrow i$
07. $DS[i, 0] \leftarrow BB[i, ord_\Sigma(x_i)]$
08. **se** $DS[i, 0] = i$ **então**
09. $Vért[i, 0].Rótulo \leftarrow DS[i, 0]$
10. $Vért[i, 0].Pai \leftarrow v_0$
11. **para** $q = 0$ até $\log n - 1$ **faça**
12. **para** $1 \leq i \leq n$ **faça-em-paralelo**
13. $k_1 \leftarrow DS[i, q]$
14. $k_2 \leftarrow DS[i + 2^q, q]$
15. $BB[k_1, k_2] \leftarrow i$
16. $DS[i, q + 1] \leftarrow BB[k_1, k_2]$
17. $Adj[k_1] \leftarrow i$
18. **para** $i \in \{j : DS[j, q] \neq j \text{ e } 1 \leq j \leq n\}$ **faça-em-paralelo**
19. $k_1 \leftarrow DS[i, q]$
20. $Adj[k_1] \leftarrow \times$
21. **para** $1 \leq i \leq n$ **faça-em-paralelo**
22. **se** $DS[i, q + 1] = i$ **então**
23. $Vért[i, q + 1].Rótulo \leftarrow DS[i, q + 1]$
24. $Vért[i, q + 1].Pai \leftarrow (DS[i, q], q)$
25. **se** $Adj[DS[i, q]] = i$ **então**
26. $Vért[i, q + 1].Pai \leftarrow Vért[DS[i, q], q].Pai$
27. $Vért[DS[i, q], q] \leftarrow \lambda$
28. **Fim**

A atribuição de nomes é realizada para que os vértices da árvore esqueleto, e posteriormente da árvore dos sufixos, possam armazenar descritores de subpalavras ao invés de subpalavras inteiras. Note que o espaço necessário para armazenamento de um descritor, utilizando o critério de custo uniforme, é constante, enquanto que o espaço necessário para armazenar uma subpalavra de $x\#$ é, no pior caso, proporcional a n .

A principal idéia da atribuição concorrente de nomes ao conjunto dos fatores básicos de $x\#$ é ilustrada abaixo. Supondo que a atribuição de nomes a todas as subpalavras de x , digamos de tamanho 2^q , já tenha sido realizada, ou seja, que a iteração $q-1$, onde $0 < q < \log n$, já tenha sido executada, a atribuição de um nome à subpalavra $x_i x_{i+1} \dots x_{i+2^{q+1}-1}$ de $x\#$ que inicia na posição i , para todo $1 \leq i \leq n$, e tem tamanho 2^{q+1} , ocorre na iteração seguinte q , da seguinte forma. Os valores $DS[i, q]$ e $DS[i+2^q, q]$, previamente computados, serão as coordenadas para a matriz BB , onde tenta-se escrever o valor i . Observe que, caso exista alguma subpalavra $x_j x_{j+1} \dots x_{j+2^{q+1}-1}$ de $x\#$, idêntica à $x_i x_{i+1} \dots x_{i+2^{q+1}-1}$, para algum $j, 1 \leq j \neq i \leq n$, então temos que $DS[i, q] = DS[j, q] = k_1$ e $DS[i+2^q, q] = DS[j+2^q, q] = k_2$. Isso significa que, neste caso, ocorrerá uma concorrência na escrita dos valores i e j em $BB[k_1, k_2]$. Supomos que o menor valor terá prioridade. Em seguida, o conteúdo dessa posição de BB será atribuído a $DS[i, q+1]$ e $DS[j, q+1]$, caracterizando a atribuição de um único nome às subpalavras $x_i x_{i+1} \dots x_{i+2^{q+1}-1}$ e $x_j x_{j+1} \dots x_{j+2^{q+1}-1}$ de $x\#$.

$$x\# = x_1 \dots \underbrace{x_i \dots x_{i+2^q-1}}_{DS[i, q]} \underbrace{x_{i+2^q} \dots x_{i+2^{q+1}-1}}_{DS[i+2^q, q]} \dots \underbrace{x_j \dots x_{j+2^q-1}}_{DS[j, q]} \underbrace{x_{j+2^q} \dots x_{j+2^{q+1}-1}}_{DS[j+2^q, q]} \dots x_n \#^{n-1}$$

No processo de criação de vértices, em uma dada iteração, precisamos verificar a existência de adjacências, para evitar a ocorrência de vértices com um único filho em D_x . Essa verificação é feita da seguinte forma. No mesmo exemplo da iteração q , seja a subpalavra $x_i \dots x_{i+2^q-1}$. Suponha que o tamanho do conjunto $S_{x_i \dots x_{i+2^q-1}}$ das subpalavras de tamanho $2 \cdot |x_i \dots x_{i+2^q-1}|$ e que têm $x_i \dots x_{i+2^q-1}$ como prefixo, seja maior que 1, ou seja, suponha que exista mais que uma palavra de tamanho 2^{q+1} que tem $x_i \dots x_{i+2^q-1}$ como prefixo. Então, no procedimento *Árvore Esqueleto*, na linha $DS[i, q] = k_1$ da matriz BB , com $1 \leq k_1 \leq n$, serão escritos mais que um valores em diferentes colunas, referentes a essas subpalavras em $S_{x_i \dots x_{i+2^q-1}}$. Isto indica a existência de vértices adjacentes sendo criados. Caso contrário, isto é, se $S_{x_i \dots x_{i+2^q-1}}$ tem tamanho igual a 1, o vértice no nível q que está rotulado com a primeira componente do descritor associado à $x_i \dots x_{i+2^q-1}$, será substituído por um vértice no nível $q+1$ rotulado com a primeira componente do descritor da subpalavra $x_i x_{i+1} \dots x_{i+2^{q+1}-1}$. Essa idéia da verificação de adjacências está implementada nas linhas de 18 até 20 do procedimento *Árvore Esqueleto*. Portanto, a cada iteração q , $0 \leq q < \log n$, os vértices do nível $q+1$ são criados, conforme a segunda parte da Definição 4.2.2. A Figura 4.3 ilustra essa idéia. Observe que a corretude desse procedimento, enunciada no Lema 4.2.3, pode ser facilmente demonstrada por indução em seu número de iterações, através da discussão que acabamos de apresentar.

Lema 4.2.3 *O procedimento Árvore Esqueleto constrói corretamente uma árvore esqueleto D_x associada a uma palavra x de tamanho n . Isto é, o procedimento Árvore Esqueleto computa corretamente as matrizes DS e $Vért$.* $\square$

Figura 4.3: Uma ilustração da idéia da atribuição concorrente de nomes às subpalavras de $x\#$ e da criação de vértices de D_x . O procedimento *Árvore Esqueleto*, que recebeu como entrada a palavra $x\# = abbabbabbabbaab\#\#\dots\#$, executando a iteração $q = 2$. Observe que $\mathcal{W}(u) = abba$ e que $S_{abba} = \{abbabbab, abbabbaa, abbaab\#\#\}$. Dessa forma, na iteração $q = 2$, além da atribuição concorrente de nomes a todas as subpalavras de tamanho 2^3 de $x\#$, ocorre a criação concorrente de vértices de nível $q + 1 = 3$. Os vértices u_1, u_2 e u_3 , filhos de u , são criados nessa iteração, de tal maneira que $\mathcal{W}(u_1) = abbabbab$, $\mathcal{W}(u_2) = abbabbaa$ e $\mathcal{W}(u_3) = abbaab\#\#\}$. Observe também que, $\mathcal{W}(v) = baab$ e que $S_{baab} = \{baab\#\#\#\}$. Assim, o vértice v é eliminado do nível $q = 2$ e um novo vértice v_1 é criado no nível $q + 1 = 3$ tal que $\mathcal{W}(v_1) = baab\#\#\#\}$.

O procedimento *Árvore Esqueleto* contém um laço seqüencial, nas linhas de 11 até 27, que implementa, a cada iteração, a atribuição concorrente de nomes, a criação concorrente de vértices e a verificação de ocorrência de vértices adjacentes que acabamos de discutir. Note que a atribuição de nomes em paralelo é realizada no laço das linhas de 12 até 17, em tempo constante, utilizando um número de operações proporcional a n . Da mesma forma, o laço nas linhas de 18 até 20, para verificação de adjacências é executado em tempo constante e o número de operações utilizadas é proporcional a n . Ainda, o laço de criação de nomes em paralelo, nas linhas de 21 até 27 também gasta tempo constante e utiliza um número de operações proporcional a n . Dessa forma, já que o laço seqüencial possui $\log n$ iterações, temos que o tempo paralelo do principal laço desse procedimento é proporcional a $\log n$ e o trabalho realizado é proporcional a $n \log n$. O procedimento possui ainda dois laços de inicializações, nas linhas de 2 até 10, sendo que o tempo paralelo gasto nessas inicializações é proporcional a $\log n$ e o trabalho realizado é proporcional a $n \log n$. Portanto, a complexidade de tempo paralelo do procedimento *Árvore Esqueleto* é $O(\log n)$ e o trabalho total realizado é $O(n \log n)$. Finalmente, observe que utilizando a matriz BB , o espaço total gasto é $O(n^2)$. Essa argumentação prova o Lema 4.2.4 a seguir.

Lema 4.2.4 *A complexidade de tempo paralelo do procedimento Árvore Esqueleto, que constrói a árvore esqueleto D_x associada a uma palavra x de tamanho n , sobre um alfabeto Σ de tamanho constante, é $T(n) = O(\log n)$, o trabalho realizado é $\tau(n) = O(n \log n)$ e o espaço utilizado é $O(n^2)$.*

□

Antes de apresentar finalmente um exemplo de construção da árvore esqueleto, observamos que a matriz BB não necessita ser inicializada durante a execução desse procedimento. Isso se deve à garantia de que uma leitura será realizada somente em posições de BB prévia e devidamente escritas.

Um exemplo de execução do procedimento *Árvore Esqueleto* D_x , que recebe como entrada a palavra $x\# = abbabbabbabbaab\#\#\dots\#$, relacionado ao exemplo da Figura 4.2, é ilustrado na Figura 4.4.

Árvore Esqueleto com Rótulos Modificados

Para iniciar a transformação da árvore esqueleto D_x na árvore dos sufixos T_x , os rótulos de todos os vértices $v \in V$ de D_x devem ser modificados. Observe que todo vértice $v \in D_x$, representado por $Vért[i, q] \neq \lambda$, para algum $1 \leq i \leq n$ e $0 \leq q \leq \log n$, contém a primeira componente do descritor da subpalavra de $x\#$ que começa na posição i e tem tamanho 2^q . Essa informação é redundante, já que $DS[i, q]$ também deve contê-la. Antes de apresentar um algoritmo que implementa tais modificações, apresentaremos a definição de rótulo modificado.

Figura 4.4: A matriz dos descritores DS e a matriz dos vértices $Vért$ de D_x , resultadas da execução do procedimento *Árvore Esqueleto* recebendo como entrada a palavra $x = abbabbabbabbaab\#\#\dots\#$. Na possibilidade de ocorrência de uma escrita concorrente, a convenção adotada neste exemplo é de que o menor valor terá prioridade. A figura também apresenta as matrizes BB e Adj no final da iteração 3.

Definição 4.2.5 *Seja $v \in V$ um vértice da árvore esqueleto D_x tal que o Rótulo de v contém a posição de início de $\mathcal{W}(v)$ em $x\#$. O rótulo modificado ou m-rótulo de v é um par (i, l) tal que, se $\mathcal{W}(v) = \mathcal{W}(\mathcal{P}(v)) \cdot w$, para alguma subpalavra w de $x\#$, então $l = |w|$ e i é a posição de início de uma ocorrência de w em $x\#$.*

Descreveremos agora um procedimento que transforma a árvore esqueleto D_x , construída anteriormente, na **árvore esqueleto m-rotulada** $m-D_x$ ou na **árvore esqueleto com rótulos modificados** $m-D_x$. Não considerando a orientação das arestas, a única diferença entre a árvore esqueleto com rótulos modificados $m-D_x$ e a árvore dos sufixos T_x é que não podem existir vértices com o mesmo pai em T_x tais que seus rótulos descrevam duas subpalavras de $x\#$ que possuem um prefixo em comum. Entretanto, a árvore $m-D_x$ compartilha com T_x as propriedades (1), (3) e (4) da Definição 3.1.1 de T_x , sempre observando que aqui a palavra $x\#$ está sendo utilizada no lugar da palavra x .

Um m-rótulo ou rótulo modificado de um vértice $v \in V$ de D_x pode ser trivialmente computado em tempo constante conhecendo-se o Rótulo de v e os níveis, digamos q e q' , de v e de $\mathcal{P}(v)$, respectivamente. Mais precisamente, se j é o Rótulo de v , então $(j + 2^{q'}, 2^q - 2^{q'})$ é o rótulo modificado de v . Para um vértice $Vért[i, q]$ representado na matriz $Vért$, isto é, para um vértice $Vért[i, q] \neq \lambda$, com $1 \leq i \leq n$ e $0 \leq q \leq \log n$, temos que a informação sobre o rótulo deste vértice está armazenada em $Vért[i, q].Rótulo$, que o nível deste vértice é q e que o nível q' de seu vértice pai é a segunda coordenada da informação armazenada em $Vért[i, q].Pai$. O procedimento *Árvore Esqueleto Modificada* abaixo, proposto por nós, implementa essa idéia esboçada em [AIL⁺88].

Procedimento *Árvore Esqueleto Modificada*

Entrada: a matriz $Vért$ obtida no procedimento *Árvore Esqueleto*, representando D_x .
Saída: a matriz $Vért$ preenchida com os rótulos modificados, representando $m-D_x$.

01. **Início**
02. para $1 \leq i \leq n$ faça-em-paralelo
03. $q_i \leftarrow \log n$
04. para $j = 1$ até $\log n + 1$ faça
05. para $1 \leq i \leq n$ faça-em-paralelo
06. se $Vért[i, q_i] \neq \lambda$ então
07. $k \leftarrow Vért[i, q_i].Rótulo$
08. $q' \leftarrow -1$
09. se $Vért[i, q_i].Rótulo \neq v_0$ então
10. $(s, q') \leftarrow Vért[i, q_i].Pai$
11. se $j \geq q_i - q'$ então
12. $Vért[i, q_i].Rótulo \leftarrow (k + \lfloor 2^{q'} \rfloor, 2^{q_i} - \lfloor 2^{q'} \rfloor)$
13. $q_i \leftarrow q'$
14. **Fim**

A atribuição de rótulos modificados aos vértices de D_x segue o seguinte princípio: todos os vértices, cujos vértices pais se encontram no mesmo nível, terão seus rótulos modificados. Assim, na primeira iteração por exemplo, todos os vértices folha do nível $q_i = \log n$, com $1 \leq i \leq n$, cujos vértices pais se encontram no nível $q - 1$, terão seus rótulos modificados da forma como descrevemos acima. Em seguida, na próxima iteração do procedimento, todos os vértices internos ou folhas, cujos vértices pais se encontram no mesmo nível $q_i - 2$, terão seus rótulos modificados.

Essa argumentação, feita de forma indutiva, prova a corretude do procedimento *Árvore Esqueleto Modificada*. Além disso, o procedimento contém um laço seqüencial tal que cada uma das $\log n + 1$ iterações executa um número de operações paralelas proporcional a n em tempo constante. Dessa forma, sua complexidade de tempo paralelo é $O(\log n)$ e o trabalho realizado é $O(n \log n)$. O Lema 4.2.6 abaixo pode ser demonstrado através da discussão que acabamos de apresentar.

Lema 4.2.6 *O procedimento Árvore Esqueleto Modificada computa corretamente os rótulos modificados de cada vértice $v \in V$ de D_x . A transformação de D_x em $m\text{-}D_x$ é executada em tempo paralelo $O(\log n)$, utilizando um total de $O(n \log n)$ operações.* $\square$

A Figura 4.5 mostra a matriz $Vért$ resultante do procedimento *Árvore Esqueleto Modificada*, com os rótulos dos vértices modificados.

Finalizando essa seção, vamos enunciar duas propriedades importantes relativas à árvore esqueleto modificada $m\text{-}D_x$, contidas em [AIL⁺88]. Seja (i, l) e (j, m) os rótulos modificados de dois vértices $u, v \in V$ de $m\text{-}D_x$ tais que $\mathcal{P}(u) = \mathcal{P}(v)$. Seja q o nível de $\mathcal{P}(u) = \mathcal{P}(v)$.

Proposição 4.2.7 *As subpalavras de $x\#$ cujos descritores são rótulos modificados de u e v têm um prefixo comum de tamanho não maior que $2^q - 1$.*

Demonstração.

Segue imediatamente da Definição 4.2.2 de D_x . $\square$

Proposição 4.2.8 *Se k é o tamanho do maior prefixo comum de $x_i x_{i+1} \dots x_{i+l-1}$ e $x_j x_{j+1} \dots x_{j+m-1}$, então $DS[i, \lfloor \log k \rfloor] = DS[j, \lfloor \log k \rfloor]$.*

Demonstração.

Segue imediatamente da construção da matriz DS dos descritores dos fatores básicos de $x\#$. $\square$

		<i>Vért</i>				
		0	1	2	3	4
<i>Rótulo</i> <i>Pai</i>	1	(1,1)	(2,1)	(3,2)	(5,4)	(9,8)
		v_0	(1,0)	(1,1)	(1,2)	(1,3)
2		(2,1)	(3,1)	(4,2)	(6,4)	(10,8)
		v_0	(2,0)	(2,1)	(2,2)	(2,3)
3			(4,1)	(5,2)	(7,4)	(11,8)
			(2,0)	(3,1)	(3,2)	(3,3)
4						(12,8)
						(1,3)
5						(13,8)
						(2,3)
6						(14,8)
						(3,3)
7						(11,12)
						(1,2)
8						(12,12)
						(2,2)
9						(13,12)
						(3,2)
10						(14,12)
						(1,2)
11						(13,14)
						(2,1)
12						(14,14)
						(3,1)
13						(14,15)
						(1,0)
14						(16,14)
						(1,1)
15						(16,15)
						(2,0)
16						(16,16)
						v_0

Figura 4.5: A matriz *Vért*, resultado da execução do procedimento *Árvore Esqueleto Modificada* representando a árvore esqueleto modificada $m-D_x$. O procedimento recebeu como entrada a matriz *Vért* representando a árvore esqueleto D_x associada a palavra $x\# = \text{abbabbabbabbaab}\#\#\dots\#$.

Refinamentos Sucessivos

Após a árvore esqueleto modificada $m-D_x$ ter sido construída, o próximo passo do **algoritmo AILSV** é produzir uma árvore muito semelhante à árvore dos sufixos T_x , a menos da orientação de suas arestas. Essa árvore é resultante de $\log n - 1$ refinamentos consecutivos de $m-D_x$, que chamaremos agora de $D^{(\log n-1)}$. O q -ésimo refinamento é denotado por $D^{(\log n-q-1)}$, onde $1 \leq q \leq \log n - 1$. Informalmente, $D^{(\log n-q-1)}$ é uma árvore rotulada, com n folhas, sem nenhum vértice de grau 1, e que possui uma estrutura equivalente à estrutura de $m-D_x$. Em particular, as propriedades (1), (3) e (4) da Definição 3.1.1 de T_x são verificadas em qualquer refinamento de $m-D_x$. A árvore $D^{(0)}$, produzida no refinamento $\log n - 1$, é idêntica à árvore dos sufixos T_x , exceto pela orientação de suas arestas. Ilustrativamente, os refinamentos sucessivos da árvore esqueleto modificada $m-D_x$ ocorrem da seguinte forma:

$$m-D_x = D^{(\log n-1)} \longrightarrow D^{(\log n-2)} \longrightarrow \dots \longrightarrow D^{(1)} \longrightarrow D^{(0)} \approx T_x$$

Definições mais rigorosas para os refinamentos $D^{(\log n-q-1)}$, onde $q = 1, 2, \dots, \log n - 1$, contidas em [AIL⁺88], serão apresentadas a seguir. Isso será feito através da especificação precisa de como $D^{(\log n-q-1)}$ é obtida a partir de $D^{(\log n-(q-1)-1)}$, para $q = 1, 2, \dots, \log n - 1$.

Definição 4.2.9 *Um ninho é qualquer conjunto formado pelos filhos de algum vértice de $D^{(k)}$, para todo $k = \log n - 1, \dots, 1, 0$.*

Definição 4.2.10 *Sejam (i, l) e (j, m) os rótulos de dois vértices em algum ninho de $D^{(k)}$, com $0 \leq k \leq \log n - 1$. Um inteiro t , $0 < t \leq \min\{l, m\}$, é um **refinador** para (i, l) e (j, m) se e somente se $x_i x_{i+1} \dots x_{i+t-1} = x_j x_{j+1} \dots x_{j+t-1}$.*

Suponha agora que os refinamentos anteriores a $D^{(k)}$, com $0 < k \leq \log n - 1$, já tenham sido produzidos e que $D^{(k)}$ satisfaz a seguinte **condição**(k):

1. $D^{(k)}$ é uma árvore rotulada, com n folhas e nenhum vértice de grau 1;
2. $D^{(k)}$ satisfaz as propriedades (1), (3) e (4) da Definição 3.1.1 de T_x ;
3. $D^{(k)}$ é rotulada de tal forma que nenhum par de rótulos de vértices no mesmo ninho admite um refinador de tamanho 2^k .

Observe que a **condição**(k) é trivialmente verificada para a árvore esqueleto modificada $m-D_x = D^{(\log n-1)}$. Além disso, o item 3 da **condição**(0) implica que, revertendo a orientação de todas as arestas de $D^{(0)}$ ocorrerá a transformação de $D^{(0)}$ na árvore dos sufixos T_x .

Apresentaremos agora a definição de $D^{(k-1)}$ como sendo a árvore obtida através da transformação de $D^{(k)}$. Antes disso, a definição de ninho possivelmente refinável faz-se necessária.

Definição 4.2.11 *Um ninho possivelmente refinável de $D^{(k)}$ é um ninho cujo vértice pai pode admitir um refinador de tamanho 2^{k-1} .*

Note que, pela Proposição 4.2.7 e pela Definição 4.2.11, os únicos ninhos possivelmente refináveis em $m-D_x = D^{(\log n - 1)}$ são aqueles que se encontram no nível $\log n - 1$. Os ninhos de vértices cujos vértices pais se encontram no nível $\log n - 2$ tornam-se possivelmente refináveis no próximo refinamento. De uma forma geral, pela Proposição 4.2.7 e pela Definição 4.2.11, temos que um ninho torna-se possivelmente refinável em $D^{(k)}$, para $0 < k \leq \log n - 1$, se o pai desse ninho encontra-se no nível pelo menos k em $D^{(k)}$. Finalmente, observe que a raiz v_0 nunca será o vértice pai de algum ninho refinável.

Dessa forma, suponha que em $D^{(k)}$ exista um conjunto de vértices pais de ninhos possivelmente refináveis. Devemos observar que a identificação dos vértices pais de ninhos possivelmente refináveis depende de um esquema de alocação e escalonamento de processadores que será brevemente descrito em uma seção a seguir. Dessa forma, seja $(i_1, l_1), (i_2, l_2), \dots, (i_m, l_m)$ o conjunto de todos os rótulos dos vértices de algum ninho refinável de $D^{(k)}$. Seja $u \in V$ o vértice pai deste ninho. O refinamento de um ninho é realizado em dois passos, como descrito a seguir.

Refinamento de um ninho de $D^{(k)}$

Passo 1 Utilize as matrizes $Vért$ e DS para modificar o ninho de raiz u , como segue.

Particione os filhos de u em classes de equivalência, colocando na mesma classe todos os vértices que possuem o mesmo valor para $DS[i_j, k - 1]$, com $j = 1, 2, \dots, m$. Para cada classe não unitária que resulta, execute as seguintes operações:

1. Crie um novo vértice pai v para os vértices dessa classe e faça $\mathcal{P}(v) = u$;
2. Faça o atributo *Rótulo* de v receber o valor $(i, 2^{k-1})$, onde $i = DS[i_j, k - 1]$, para qualquer $1 \leq j \leq m$;
3. Considere cada filho de v . Para o filho de v cujo *Rótulo* atual é (i_j, l_j) , troque esse valor para $(i_j + 2^{k-1}, l_j - 2^{k-1})$.

Passo 2 Se mais que uma classe resulta da partição, então pare. Caso contrário, seja C a única classe resultante da partição. Segue da hipótese (1) da **condição(k)** de $D^{(k)}$ que C não pode ser uma classe unitária. Assim, um novo vértice pai v , como acima, foi criado para ser o vértice pai dos vértices em C , durante o passo 1. Faça v um filho do vértice pai de u e faça o atributo *Rótulo* de v receber $(i, l + 2^{k-1})$, onde (i, l) é o rótulo do vértice u .

O teorema a seguir, devido a Apostolico et. al [AIL⁺88], mostra que a definição de uma série de refinamentos $D^{(k)}$ não é ambígua.

Teorema 4.2.12 *A aplicação síncrona dos passos 1 e 2 a todos os ninhos possivelmente refináveis de $D^{(k)}$ produz a árvore que satisfaz a condição(k).*

Demonstração.

As propriedades (2) e (3) da **condição(k)** são estabelecidas facilmente para $D^{(k-1)}$. Assim, nos concentraremos na propriedade (1). Já que nenhuma folha nova foi inserida na transição $D^{(k)} \rightarrow D^{(k-1)}$, a propriedade (1) será satisfeita uma vez provado que $D^{(k-1)}$ é uma árvore sem vértices de grau 1.

Como já mencionado anteriormente, o ninho dos vértices filhos da raiz v_0 não é refinável para qualquer $k > 0$. Assim, para qualquer vértice u pai de um ninho possivelmente refinável de $D^{(k)}$, o vértice $\mathcal{P}(u)$ está definido. Pela **condição(k)**, o vértice u tem mais que um filho, assim como $\mathcal{P}(u)$. Seja $\hat{D}^{(k)}$ a estrutura resultante da aplicação do Passo 1 em $D^{(k)}$.

Se, em $D^{(k)}$, o ninho de $\mathcal{P}(u)$ não é refinável, então u é um vértice de $D^{(k-1)}$, e u pode ser o único vértice de grau 1 em $D^{(k)}$ entre qualquer filho de u em $D^{(k)}$ e o pai de u em $D^{(k)}$. O vértice u é removido no Passo 2, a menos que u seja um vértice de grau > 1 em $\hat{D}^{(k)}$. Portanto, nenhum vértice de grau 1 é criado nesta parte de $D^{(k-1)}$.

Suponha agora que, em $D^{(k)}$, ambos os ninhos de u e de $\mathcal{P}(u)$ são possivelmente refináveis. Afirmamos que, em $\hat{D}^{(k)}$, ou o pai de u não foi atualizado e é um vértice de grau > 1 , ou ele foi atualizado mas ainda possui grau > 1 . De fato, pela definição de $D^{(k)}$, ou o ninho de u ou o ninho de $\mathcal{P}(u)$ não será refinado em uma classe de equivalência unitária. Assim, no final do Passo 1, tem-se as seguintes alternativas:

1. O pai de u em $\hat{D}^{(k)}$ é idêntico $\mathcal{P}(u)$ em $D^{(k)}$. Como o ninho de $\mathcal{P}(u)$ não poderia ter sido refinado em um classe unitária, então $\mathcal{P}(u)$ deve ser um vértice de grau > 1 em $D^{(k-1)}$. Assim este caso se reduz àquele onde o ninho de $\mathcal{P}(u)$ não é refinável;
2. O pai de u em $\hat{D}^{(k)}$ não é o pai de u em $D^{(k)}$. Então o $\mathcal{P}(u)$ em $\hat{D}^{(k)}$ é um vértice de grau > 1 , e também um vértice de $D^{(k-1)}$. Se u é um vértice de grau > 1 em $\hat{D}^{(k)}$, então não existe vértice de grau 1 entre u e $\mathcal{P}(u)$ em $\hat{D}^{(k)}$, e o mesmo é verdadeiro entre qualquer vértice no ninho de u e u . Se u é um vértice de grau 1 em $\hat{D}^{(k)}$, então o único filho de u é um vértice de grau > 1 . Como o pai atual de u é também um vértice de grau > 1 por hipótese, então removendo u no Passo 2 elimina-se o único vértice de grau 1 existente no caminho de qualquer vértice no ninho de u para o mais próximo ancestral de grau > 1 desse vértice.

□

Os refinamentos sucessivos descritos acima foram aplicados sobre a árvore esqueleto modificada $m-D_x$, associada à palavra $x\# = abbabbabbabbaab\#\#\dots\#$ do nosso exemplo da Figura 4.5, resultando a matriz $Vért$ da Figura 4.7 que representa a árvore $D^{(0)}$ da Figura 4.6.

Figura 4.6: A árvore $D^{(0)}$ obtida dos refinamentos sucessivos da árvore esqueleto modificada $m-D_x$ associada à palavra $x\# = abbabbabbabbaab\#\#\dots\#$. Note que a árvore $D^{(0)}$ é a árvore dos sufixos T_x , a menos da orientação de suas arestas.

A corretude do procedimento que transforma a árvore esqueleto modificada $m-D_x$ na árvore $D^{(0)}$ pode ser demonstrada através do Teorema 4.2.12, por indução no número k de refinamentos.

Lema 4.2.13 *Os $\log n - 1$ refinamentos consecutivos de todos os ninhos possivelmente refináveis da árvore esqueleto modificada $m-D_x$ realizam corretamente a transformação de $m-D_x$ na árvore $D^{(0)}$.*

□

Como os refinamentos realizados sobre os ninhos refináveis de uma dada iteração são executados em paralelo em tempo constante, então o tempo total desse procedimento é $O(\log n)$ e o número total de operações efetuadas é $O(n \log n)$.

		<i>Vért</i>				
		0	1	2	3	4
<i>Rótulo</i>	1	(1,1)	(2,1)	(3,2)	(8,3)	(11,6)
<i>Pai</i>		v_0	(1,0)	(1,1)	(7,3)	(1,3)
	2	(2,1)	(3,2)	(5,3)	(8,3)	(11,7)
		v_0	(2,0)	(2,1)	(2,2)	(2,3)
	3		(4,1)	(5,2)	(8,3)	(11,8)
			(2,0)	(3,1)	(3,2)	(3,3)
	4					(14,6)
						(1,3)
	5					(14,7)
						(2,3)
	6					(14,8)
						(3,3)
	7				(2,3)	(14,9)
					(1,2)	(7,3)
	8					(14,10)
						(2,2)
	9					(14,11)
						(3,2)
	10					(14,12)
						(1,2)
	11					(14,13)
						(2,1)
	12					(14,14)
						(3,1)
	13					(14,15)
						(1,0)
	14					(16,14)
						(1,1)
	15					(16,15)
						(2,0)
	16					(16,16)
						v_0

Figura 4.7: O procedimento *Refinamentos Sucessivos* recebe como entrada a matriz *Vért* representando a árvore esqueleto modificada $m-D_x$ associada a palavra $x\# = \text{abbabbabbabbaab}\#\#\dots\#$. A figura mostra a matriz *Vért*, resultado dos refinamentos sucessivos $m-D_x = D^{(3)} \rightarrow D^{(2)} \rightarrow D^{(1)} \rightarrow D^{(0)}$.

Lema 4.2.14 *A transformação da árvore esqueleto modificada $m\text{-}D_x$ na árvore $D^{(0)}$, através dos refinamentos sucessivos, é realizada em tempo paralelo $O(\log n)$ e o trabalho realizado é $O(n \log n)$.*

□

Revertendo as arestas de $D^{(0)}$

Em algumas aplicações, a árvore dos sufixos T_x necessita somente ser percorrida ascendente. Assim, a estrutura obtida na árvore $D^{(0)}$ é suficiente para executar essas tarefas. Entretanto, como qualquer árvore *trie*, T_x é usualmente empregada para executar buscas descendentes, começando de sua raiz v_0 . Isto requer a inclusão, para cada aresta ascendente original $v \rightarrow u$, de uma aresta descendente similar $u \rightarrow v$. Correspondentemente, cada vértice u deve agora armazenar rótulos descendentes apropriados para toda aresta descendente que sai de u . Tais rótulos fornecem as informações das ramificações necessárias no curso de uma busca descendente de uma palavra w , sobre Σ , na árvore dos sufixos T_x .

Existem basicamente duas maneiras de se definir essas informações de ramificação, descritas em [AIL⁺88]. A primeira forma são as informações de ramificação baseadas diretamente nos símbolos de Σ e a segunda, são informações de ramificação baseadas nos descritores dos símbolos iniciais das subpalavras de $x\#$, que rotulam as arestas da árvore $D^{(0)}$. Seja (i, l) o rótulo de uma aresta ascendente $v \rightarrow u$ de $D^{(0)}$, armazenado em v . Na primeira forma deve-se rotular a aresta descendente $u \rightarrow v$ com o símbolo de Σ que corresponde a x_i . O conjunto de rótulos descendentes do vértice interno $u \in V$ de $D^{(0)}$ pode então ser armazenado em uma lista linear, uma árvore *trie* binária ou um vetor. A implementação que acabamos de apresentar é mais eficiente no espaço gasto e menos eficiente na identificação de uma ramificação, comparando-a com a implementação que tentaremos descrever mais cuidadosamente a seguir.

A segunda maneira de definir informações de ramificação para o vértice u é baseada na utilização do valor $DS[i, 0]$, isto é, do nome atribuído ao símbolo x_i de x . A implementação de rótulos descendentes em vetores, baseada na matriz DS , é bastante simples e utilizada na maioria das aplicações também descritas em [AIL⁺88]. Como veremos adiante, o espaço gasto nessa implementação é $O(n^2)$ e a identificação de uma ramificação é feita em tempo constante.

Para cada vértice interno $u \in V$ de $D^{(0)}$ é atribuído um vetor de tamanho n , chamado R_u . O vetor R_u armazena as ramificações, ou as arestas descendentes de u , da seguinte forma. Se o vértice $v \in V$ de $D^{(0)}$ é tal que $\mathcal{P}(v) = u$ e o Rótulo ascendente contido em v é (i, l) , então um apontador para v é armazenado em $R_u[DS[i, 0]]$, conforme ilustrado na Figura 4.8 a seguir.

Figura 4.8: A idéia da representação do vetor de ramificações R_u do vértice u , implementando a aresta $u \rightarrow v$. Se (i, l) é o descritor da aresta $v \rightarrow u$, armazenado no vértice v , cuja representação encontra-se na posição (j, q) da matriz $Vért$, isto é, tal que $Vért[j, q].Rótulo = (i, l)$, então fazemos $R_u[DS[i, 0]] \leftarrow (j, q)$.

A implementação dos vetores de ramificações dos vértices internos da árvore $D^{(0)}$ é descrita em um procedimento em alto nível, proposto por nós. O procedimento *Ramificações* recebe como entrada a árvore $D^{(0)}$ e constrói, para cada vértice interno da árvore $D^{(0)}$, os vetores de ramificações, transformando a árvore $D^{(0)}$ na árvore dos sufixos T_x , conforme a discussão que acabamos de apresentar.

Procedimento *Ramificações*

Entrada: a árvore $D^{(0)}$.

Saída: os vetores de ramificação R_u , para todo vértice interno de $D^{(0)}$, que, juntamente com a matriz $Vért$, representam a árvore dos sufixos T_x .

01. Início

02. **para** todo vértice interno $u \in V$ de $D^{(0)}$ **faça-em-paralelo**

03. **para** todo vértice $v \in V$ de $D^{(0)}$ tal que $\mathcal{P}(v) = u$ **faça**

04. Seja $Vért[j, q].Rótulo = (i, l)$ o rótulo da aresta descendente $v \rightarrow u$, armazenado em v . Então, faça $R_u[DS[i, 0]] \leftarrow (j, q)$

05. Fim

Não é difícil argumentar que o procedimento *Ramificações* pode ser implementado em tempo constante, utilizando n operações. Através de um esquema de alocação e escalonamento de processadores que tentaremos descrever na última seção do **algoritmo AILSV** que estamos apresentando, essa argumentação se torna mais clara.

Observe finalmente que as $O(n^2)$ localizações de todos os vetores R construídos no procedimento *Ramificações* não são inicializadas. A coleção final de vetores de ramificações R descreve uma árvore orientada representando T_x e mais informações irrelevantes (“lixo”). Isso significa que, em uma busca descendente, devemos necessariamente verificar a validade das informações armazenadas nos vetores de ramificações relacionados ao percurso desse caminho, na árvore dos sufixos T_x .

O Lema 4.2.15 a seguir formaliza essas discussões sobre a corretude e as complexidades deste procedimento.

Lema 4.2.15 *O procedimento Ramificações computa corretamente os vetores de ramificações de todos os vértices internos da árvore $D^{(0)}$ em tempo paralelo constante, utilizando $O(n)$ operações e utilizando espaço $O(n^2)$.*

□

Dessa forma, podemos agora estabelecer todas as complexidades gerais do **algoritmo AILSV**, já que todos os procedimentos para a construção da árvore dos sufixos T_x já foram descritos. O Teorema 4.2.16 a seguir afirma que o **algoritmo AILSV** constrói corretamente a árvore dos sufixos T_x em tempo paralelo total $O(\log n)$, utilizando um total de $O(n \log n)$ operações e utilizando espaço total $O(n^2)$.

Teorema 4.2.16 *O algoritmo AILSV [AIL⁺88] constrói corretamente a árvore dos sufixos T_x associada à palavra x de tamanho n , sobre um alfabeto Σ , com complexidade de tempo paralelo total $O(\log n)$, realizando trabalho total $O(n \log n)$ e utilizando espaço total $O(n^2)$.*

Demonstração.

Segue imediatamente dos Lemas 4.2.3 e 4.2.4, 4.2.6, 4.2.13 e 4.2.14, 4.2.15.

□

Corolário 4.2.17 *O problema de construção da árvore dos sufixos está em NC.*

□

Finalmente, escolheremos um vértice da árvore $D^{(0)}$ do nosso exemplo da Figura 4.6 para ilustrar o procedimento *Ramificações* sendo executado. A Figura 4.9 ilustra essa execução.

Figura 4.9: A implementação do vetor de ramificações do vértice u de $D^{(0)}$, representado na matriz $Vért$ por $Vért[2, 0]$. Um apontador para o vetor de ramificações é adicionado em cada célula da estrutura que compõe a matriz $Vért$. Esse novo atributo dessa estrutura é denominado *Ramif*. Observe que o descritor $(3, 2)$ do vértice filho de u , representado na matriz $Vért$ na posição $Vért[2, 1]$, fornece a posição de início da subpalavra de $x\#$ que rotula a aresta que vai do vértice $Vért[2, 1]$ ao vértice $u = Vért[2, 0]$. Assim, temos que $DS[3, 0] = 2$ será o índice dessa aresta no vetor R_u . Finalmente, $R_u[2] \leftarrow (2, 1)$ é o endereço desse vértice filho de u na matriz $Vért$.

Reduzindo o Espaço Utilizado

Como vimos anteriormente, a construção da árvore esqueleto D_x , seus refinamentos subsequentes e a construção dos vetores de ramificações dos vértices internos de $D^{(0)}$ necessitam de espaço $O(n^2)$. O procedimento *Árvore Esqueleto* necessita de espaço $O(n^2)$ devido à matriz BB , utilizada em cada iteração para a atribuição de nomes do conjunto de fatores básicos de x . Em qualquer refinamento, o ninho de cada vértice v necessita de um vetor de n posições para particionar os vértices desse ninho em classes de equivalência. Isso fica implícito na rotina de refinamento apresentada anteriormente e ficará mais claro na próxima seção. Finalmente, na transformação da árvore $D^{(0)}$ na árvore dos sufixos T_x , a cada vértice interno u é atribuído um vetor de ramificações R_u de n posições. Nesta seção apresentaremos uma forma alternativa de descrição desses processos, que necessita somente de espaço $O(n^{1+\delta})$, para qualquer $0 < \delta \leq 1$, à custa de um fator de perda no desempenho de tempo proporcional a $1/\delta$.

Analisaremos inicialmente o procedimento *Árvore Esqueleto*. A idéia a seguir é facilmente estendida aos outros procedimentos. Considere, assim, alguma subpalavra w de $x\#$ de tamanho 2^q , com $q > 0$, e seja $w = w_1 w_2$, com $|w_1| = |w_2| = 2^{q-1}$. Sejam k_1 e k_2 os nomes atribuídos por esse procedimento a w_1 e w_2 , armazenados em DS , respectivamente. Como k_1 e k_2 são inteiros entre 1 e n , existe a dificuldade de economia de espaço na criação de um nome para w , já que o par (k_1, k_2) pode assumir n^2 valores.

Apresentaremos agora uma forma de se solucionar esse problema, sugerida em [AIL⁺88] e ligeiramente modificada por nós, utilizando espaço $O(n^{1+\delta})$, para qualquer $0 < \delta \leq 1$. Focalizaremos a atribuição de um nome à palavra w , já que as mesmas manipulações serão executadas em paralelo para todas as subpalavras de x de tamanho 2^q . A idéia central deste procedimento é expressar a segunda coordenada k_2 através da sua representação na base $[n^\delta]$. Os coeficientes $(c_{1/\delta}, \dots, c_1, c_0)$ são computados em $1/\delta + 1$ passos. O procedimento *Árvore Esqueleto* é então modificado, substituindo-se as linhas de 11 até 17 originais, pela seguinte seqüência de comandos:

Procedimento <i>Árvore Esqueleto</i>

```

.
.
.
11.  para  $q = 0$  até  $\log n - 1$  faça
12.    para  $1 \leq i \leq n$  faça-em-paralelo
13.       $k_1 \leftarrow DS[i, q]$ 
14.       $k_2 \leftarrow DS[i + 2^q, q]$ 
15.       $j = 0$ 
16.      faça
17.         $c_j \leftarrow k_2 \bmod n^\delta$ 
18.         $k_2 \leftarrow \lfloor k_2 / n^\delta \rfloor$ 
19.         $BB[k_1, c_j] \leftarrow i$ 
20.         $k_1 \leftarrow BB[k_1, c_j]$ 
21.         $j++$ 
22.      até que  $j > 1/\delta + 1$ 
23.       $DS[i, q + 1] \leftarrow k_1$ 
24.       $Adj[k_1] \leftarrow i$ 
.
.
.
```

Dessa forma, a iteração q do procedimento *Árvore Esqueleto* é agora modificada para conter $1/\delta + 1$ sub-iterações como acima. A entrada para a sub-iteração $j = 0, \dots, 1/\delta$, é a seguinte:

1. Um descritor para o par consistindo da subpalavra esquerda e a $(j - 1)$ -tupla $(c_{j-1}, \dots, c_0)$. Este descritor é um número entre 1 e n ;
2. O descritor de valor c_j , ou seja, um número entre 0 e $n^\delta - 1$.

A saída da sub-iteração j é um descritor para o par consistindo da subpalavra esquerda e a j -tupla $(c_j, \dots, c_0)$. Este descritor é um número entre 1 e n .

As escritas concorrentes que possivelmente ocorrerão em qualquer sub-iteração da iteração q do procedimento *Árvore Esqueleto* modificado são similares às escritas concorrentes que possivelmente ocorrem sem tais modificações. A única diferença é que, agora, uma matriz auxiliar BB de tamanho $n \times n^\delta$ é suficiente. Para qualquer δ fixo, com $0 < \delta \leq 1$, o espaço total necessário é limitado por $O(n^{1+\delta})$ e o tempo de execução por $O((1/\delta) \log n)$.

Os refinamentos sucessivos e a inversão das arestas de $D^{(0)}$, isto é, a criação dos vetores de ramificação dos vértices internos de $D^{(0)}$, podem utilizar idéias semelhantes para reduzir o espaço utilizado.

Implementação no Modelo PRAM

O **algoritmo AILSV** para construção da árvore dos sufixos T_x , associada a uma palavra x de tamanho n , é descrito por Apostolico et. al [AIL⁺88] no modelo PRAM CRCW arbitrário, com a utilização de $O(n)$ processadores, diferentemente da descrição na Estrutura Trabalho-Tempo de Apresentação de Algoritmos Paralelos, apresentada anteriormente por nós.

Dessa forma, gostaríamos de finalizar a descrição deste algoritmo apresentando concisamente apenas as idéias básicas das principais tarefas realizadas pelos n processadores, já que o estudo detalhado do esquema de alocação e escalonamento de tais processadores é de interesse independente.

No início do **algoritmo AILSV**, na fase de construção da árvore esqueleto D_x , cada processador p_i é atribuído a uma linha i da matriz dos descritores DS , para todo $1 \leq i \leq n$, de tal forma que, a cada iteração q , com $0 \leq q \leq \log n - 1$, os valores $DS[i, q+1]$ são os nomes atribuídos aos fatores básicos de x que começam na posição i e têm tamanho 2^{q+1} . Caso o processador p_i tenha sucesso, isto é, seja o vencedor na atribuição do nome $DS[i, q+1]$, então p_i recebe a tarefa de criar um novo vértice correspondente na matriz dos vértices $Vért$, na posição $Vért[i, q+1]$. Os processadores que fracassam tornam-se inativos nesse passo. Dessa forma, ao final do procedimento *Árvore Esqueleto*, cada processador p_i está atribuído a uma folha i de D_x , isto é, está atribuído às posições $DS[i, \log n]$ e $Vért[i, \log n]$ da matriz dos descritores e da matriz dos vértices, respectivamente. A verificação da existência de arestas adjacentes é realizada da seguinte forma. Para cada linha l da matriz BB existe uma localização de memória distinta $Adj[l]$, conhecida por todos os processadores. A cada iteração q , com $0 \leq q \leq \log n - 1$, o vértice novo criado possui mais que um filho se e somente se dois ou mais processadores vencedores escrevem em diferentes posições da mesma linha de BB . Para saber se esse é o caso, todos os processadores que escrevem na mesma linha l de BB tentam escrever seu índice

em $Adj[l]$. Em seguida, todos os processadores desta linha, exceto aquele que conseguiu escrever seu índice em $Adj[l]$, escrevem uma marca especial $\times$ em $Adj[l]$. Finalmente, todos os processadores na mesma linha verificam o conteúdo de $Adj[l]$. Claramente, o processador p_i é o único processador que escreve na linha l se e somente se, neste último passo de verificação, $Adj[l] = i$.

A Figura 4.10 ilustra a disposição dos n processadores p_i na árvore esqueleto D_x , associada à palavra $x\# = abbabbabbabbaab\# \dots \#$, ao final do procedimento *Árvore Esqueleto*, em relação ao exemplo de construção da árvore esqueleto da Figura 4.2.

Figura 4.10: A alocação dos processadores p_i na árvore esqueleto D_x , ao final do procedimento *Árvore Esqueleto*, executado sobre a palavra $x\# = abbabbabbabbaab\# \dots \#$.

O próximo estágio do **algoritmo AILSV** é a modificação dos rótulos da árvore esqueleto D_x . Inicialmente, todos os processadores atribuídos a folhas, cujos vértices pai encontram-se no nível $\log n - 1$ de D_x modificam os rótulos contidos nestes vértices. Em seguida, os processadores competem na tentativa de, simultaneamente, subir ao vértice pai comum, sendo que somente um deles torna-se vencedor. A $(q - 1)$ -ésima iteração envolve todos os processadores sobre os vértices de nível q ou superior. Como resultado do processo de construção da árvore esqueleto com rótulos modificados $m-D_x$ temos um mapeamento que atribui algumas folhas e vértices internos a processadores de tal forma que a seguinte propriedade, ilustrada na Figura 4.11, é satisfeita.

Proposição 4.2.18 *Se um vértice interno tem k filhos, então precisamente $k - 1$ processadores estão atribuídos a $k - 1$ destes filhos. Mais que isso, cada um dos $k - 1$ processadores conhece o endereço do único vértice irmão sem processador atribuído.*

□

Figura 4.11: A alocação dos processadores p_i na árvore esquelética modificada $m-D_x$, ao final do procedimento *Árvore Esquelética Modificada*, executado sobre a árvore esquelética da Figura 4.10.

Nos refinamentos subsequentes realizados sobre a árvore $m-D_x$, se um ninho de $D^{(k)}$, de raiz u , possui uma linha de trabalho l da matriz BB como rascunho, então a transformação sofrida por este ninho no Passo 1 deste refinamento pode ser completada por m processadores em tempo constante, onde m é o número de filhos de u . Cada processador trabalha sobre um vértice filho, gerando os rótulos das subpalavras com metade do tamanho da palavra armazenada neste vértice, utilizando as matrizes DS e $Vért$. Depois disso, os processadores utilizam a linha l de BB atribuída ao ninho e mais os rótulos gerados para particioná-los em classes de equivalência: cada processador neste ninho, cuja primeira componente dos rótulos gerados é igual a i , compete para escrever o endereço de seu vértice na i -ésima localização da linha l de BB . Dessa maneira, um processador representante é eleito para cada classe de equivalência. Classes de equivalência unitárias

podem ser marcadas facilmente por uma segunda escrita concorrente restrita aos processadores perdedores (após esta segunda escrita, um processador representante que ainda lê seu próprio endereço em l , sabe que ele próprio representa uma classe unitária). Agora, os processadores representantes de cada classe não unitária criam os vértices pais novos, rotulando-os com a primeira componente dos seus rótulos gerados como acima, e fazem cada novo vértice acessível por todos os outros processadores na classe. Para concluir o Passo 1, os processadores na mesma classe atualizam os rótulos de seus vértices. No Passo 2, a existência de mais que uma classe de equivalência necessita ser testada. Isto é feito através da competição dos processadores representantes que utilizam a raiz do ninho como uma localização comum de escrita. Se somente uma classe de equivalência foi produzida no Passo 1, então seu processador representante executa o ajuste do seu rótulo descrito no Passo 2. Essa discussão sugere que, uma vez que um vértice de, digamos, $m-D_x = D^{(\log n-1)}$, é atribuído a um processador distinto, $D^{(\log n-2)}$ poderia ser produzida em tempo constante. A dificuldade, entretanto, está em como atribuir os vértices (especialmente, os novos vértices inseridos) de $D^{(\log n-2)}$ em tempo constante. Por definição, $D^{(k)}$ não possui vértices de grau 1. Parece então que as manipulações dos Passos 1 e 2 podem ser operadas em tempo constante, atribuindo $m-1$ processadores, ao invés de m , a um ninho de m vértices. A única suposição adicional a ser feita é que, no início, todos os processadores têm acesso ao único vértice que não possui processador atribuído. Antes de começar o Passo 1, os processadores elegem um processador que estará atribuído a dois vértices do ninho. Depois de cada passo elementar, este processador simulador “alcança” os outros. De acordo com a Proposição 4.2.18, isto mostra que n processadores podem obter o primeiro refinamento de D_x . Na atribuição de linhas de BB aos vértices de $D^{(k)}$, simplesmente atribui-se a i -ésima linha ao processador p_i . Então, sempre que p_i é encarregado de simular dois processadores em um ninho, sua linha em BB é utilizada por todos os processadores deste ninho.

Para qualquer dado valor de k , seja uma atribuição legítima de processadores para os vértices de $D^{(k)}$ uma atribuição que satisfaz a Propriedade 4.2.18.

Teorema 4.2.19 *Dada uma atribuição legítima de processadores para $D^{(k)}$, uma atribuição legítima de processadores para $D^{(k-1)}$ pode ser produzida em tempo constante.*

□

A Figura 4.12 ilustra a atribuição de processadores à árvore $D^{(0)}$ ao final dos refinamentos sucessivos realizados sobre $m-D_x$.

Finalmente, no processo de criação dos vetores de ramificações R de T_x , temos que n processadores atribuídos legitimamente a $D^{(0)}$, possuindo $O(n)$ localizações cada um, podem transformar $D^{(0)}$ em T_x em tempo constante.

Figura 4.12: A alocação dos processadores p_i na árvore $D^{(0)}$, ao final dos refinamentos sucessivos, executados sobre a árvore esqueleto modificada $m-D_x$ da Figura 4.11.

4.2.3 Algoritmo Sahinalp-Vishkin

Como visto na Seção 4.1, existem diversos algoritmos seqüenciais para construção da árvore dos sufixos com complexidades de tempo linear. No entanto, o número de operações do **algoritmo AILSV**, que acabamos de descrever, é proporcional a $n \log n$. Esse algoritmo é baseado na atribuição de nomes de subpalavras, equivalentemente ao algoritmo seqüencial clássico [KMR72], que realiza assintoticamente o mesmo número de operações. O **algoritmo Sahinalp-Vishkin**, proposto em [SV94], determina uma maneira de quebrar a simetria que ocorre no processo de atribuição de nomes, utilizando a técnica de **Lançamento Determinístico de Moedas** (*Deterministic Coin Tossing* — *DCT*), e assim reduz para linear o número de atribuições de nomes a subpalavras. Apresentaremos a seguir a motivação para o **Problema da Quebra de Simetria**, uma visão geral do **algoritmo Sahinalp-Vishkin** básico e as idéias envolvidas no projeto de um algoritmo ótimo no sentido fraco para a construção da árvore dos sufixos, associada a uma palavra sobre um alfabeto de tamanho constante.

A complexidade de trabalho do **algoritmo AILSV** é $\tau(n) = O(n \log n)$. A técnica utilizada em [KMR72, AIL⁺88] é ineficiente pela seguinte razão: durante a execução desses algoritmos ocorre a atribuição de nomes a todas as $n - 1$ subpalavras de tamanho 2;

em seguida, ocorre a atribuição de nomes a todas as $n - 3$ subpalavras de tamanho 4; e em geral, ocorre a atribuição de nomes a todas as $(n - 2^i + 1)$ subpalavras de tamanho 2^i , onde $1 \leq i \leq \log n$. Isto resulta em um número de nomes atribuídos proporcional a $n \log n$ e esse valor dita a complexidade de trabalho do **algoritmo AILSV**. O fator logarítmico na contagem dos nomes se deve ao incremento na redundância entre as subpalavras, já que ocorrem sobreposições, a medida que elas se tornam maiores. O grande problema encontrado então é que não existe uma maneira consistente de selecionar somente uma palavra dentro de um conjunto de palavras sobrepostas, já que elas são indistinguíveis nesse sentido. A principal idéia deste algoritmo é introduzir uma solução para este **problema da quebra de simetria**.

Apresentaremos a seguir a descrição em alto nível do algoritmo básico, apresentada em [SV94], de complexidade de tempo $O(\log^2 n)$ e complexidade de trabalho $O(n \log \log n)$, para um alfabeto cujo tamanho é polinomial em n . Uma implementação alternativa desse algoritmo, transformando-o em um algoritmo aleatorizado, tem complexidade de tempo $O(n^\delta)$, com $0 < \delta \leq 1$, e trabalho $O(n \log^* n)$. Através de algumas alterações introduzidas no algoritmo básico, obtém-se um algoritmo ótimo no sentido fraco, isto é, de complexidade de trabalho $O(n)$, e de complexidade de tempo $O(\log^2 n)$, para um alfabeto de tamanho constante.

Algoritmo Básico

Seja x uma palavra de tamanho n sobre um alfabeto Σ de tamanho polinomial em n . O algoritmo básico consiste de três estágios. No primeiro estágio ocorre a atribuição de nomes a algumas subpalavras de x , com o reconhecimento de algumas identidades. Essa atribuição é realizada de forma iterativa.

Na iteração 1, a palavra x é particionada em no máximo $n/2$ blocos. A cada bloco é atribuído um número entre 1 e n , denominado **nome**, de tal forma que satisfaça as duas seguintes **propriedades de consistência**:

Consistência da Partição: seja alguma subpalavra de x “suficientemente longa”, que começa na posição i de x e seja uma outra subpalavra de x que inicia na posição j de x , tal que as duas subpalavras são idênticas. Então, com exceção de possíveis margens esquerda e direita, essas subpalavras são particionadas da mesma maneira.

Consistência da Atribuição: todos os blocos consistindo da mesma subpalavra de x recebem o mesmo nome.

As propriedades de consistência foram introduzidas acima informalmente. A Figura 4.13 a seguir ilustra essas propriedades.

Figura 4.13: Visualização das propriedades de consistência: consistência da partição, consistência da atribuição e margens.

Dessa forma, a iteração 1 particiona $x(0) = x$, “reduzindo-a” à palavra $x(1)$; o tamanho de $x(1)$ é no máximo metade do tamanho de $x(0)$. Iterações subseqüentes realizam o mesmo procedimento. A iteração i , com $i = 2, 3, \dots$, reduz a subpalavra $x(i-1)$ à subpalavra $x(i)$, satisfazendo propriedades de consistência e atribuição similares. O tamanho de $x(i)$ é no máximo $n/2^i$.

Os nomes dos blocos obtidos na iteração i são chamados *i*-rótulos, e as subpalavras que eles representam são chamadas *i*-subpalavras. Observe que um *i*-rótulo é um nome que representa uma *i*-subpalavra. Este rótulo é considerado como um *i*-símbolo em $x(i)$, que será a entrada para $(i+1)$ -ésima iteração do primeiro estágio. As $(i-1)$ -subpalavras são chamadas **construtoras** das *i*-subpalavras.

Para motivar o segundo estágio, observe que as palavras $x(i)$, com $i = 0, 1, \dots$ fornecem um sistema hierárquico de subconjuntos sobre o conjunto dos índices $\{1, 2, \dots, n\}$, representando partições cada vez maiores de $\{1, 2, \dots, n\}$. Especificamente, $x(0)$ fornece subconjuntos unitários onde cada subconjunto contém um único índice. Os subconjuntos de $x(i)$ satisfazem as seguintes propriedades: (i) cada subconjunto de $x(i)$ é resultado da união de subconjuntos de $x(i-1)$; (ii) a união dos subconjuntos de $x(i)$ é o conjunto $\{1, 2, \dots, n\}$ e (iii) os subconjuntos são disjuntos, isto é, a interseção de qualquer par de subconjuntos é vazia.

O **algoritmo Sahinalp-Vishkin** básico constrói a árvore dos sufixos associada à palavra x em iterações. Entretanto, a propriedade de consistência da partição é muito fraca, já que não se pode garantir que quaisquer duas subpalavras idênticas de x são particionadas, e portanto rotuladas, da mesma forma. O segundo estágio desenvolve um sistema hierárquico de subconjuntos alternativo, detalhado a seguir. Um subconjunto nesse sistema alternativo é denominado **núcleo**. No contexto de núcleos, a letra η é utilizada para denotar subpalavras.

Semelhantemente ao primeiro estágio, $\eta(0) = x(0) = x$ e os núcleos, ou subconjuntos, de $\eta(i)$, com $i = 1, 2, \dots$ satisfazem: (i) cada núcleo de $\eta(i)$ é a união dos núcleos de $\eta(i-1)$ e (ii) a união dos núcleos de $\eta(i)$ é o conjunto $\{1, 2, \dots, n\}$. Entretanto, (iii) a interseção de alguns pares de núcleos de $\eta(i)$ não é vazia.

Núcleos representando a mesma subpalavra serão atribuídos com o mesmo nome. O exemplo a seguir tenta esclarecer a relação entre as subpalavras, os núcleos e os nomes dos núcleos.

Exemplo. Seja *abacadebdcaca* uma subpalavra de x tal que $x_{11} = a, x_{12} = b, x_{13} = a, x_{14} = c, x_{15} = a, x_{16} = d, x_{17} = e, x_{18} = b, x_{19} = d, x_{20} = c, x_{21} = a, x_{22} = c, x_{23} = a$. Suponha que $\{11, 12, 13, 14, 15, 16, 17, 18\}$, $\{14, 15, 16, 17, 18, 19\}$, e $\{16, 17, 18, 19, 20, 21, 22, 23\}$ são núcleos de $\eta(1)$. Sejam 3, 2 e 6 os nomes desses núcleos, respectivamente. Então, em $\eta(1)$ uma subpalavra 3 2 6 corresponde à subpalavra acima de x . Agora, seja *abacadebacaca* uma subpalavra de x tal que $x_{31} = a, x_{32} = b, x_{33} = a, x_{34} = c, x_{35} = a, x_{36} = d, x_{37} = e, x_{38} = b, x_{39} = a, x_{40} = c, x_{41} = a, x_{42} = c, x_{43} = a$, e suponha que $\{31, 32, 33, 34, 35, 36, 37, 38\}$, $\{33, 34, 35, 36, 37, 38, 39\}$ e $\{36, 37, 38, 39, 40, 41, 42, 43\}$ são núcleos de $\eta(1)$. Sejam os nomes desses núcleos 3, 1 e 5, respectivamente. Então, em $\eta(1)$ uma subpalavra 3 1 5 corresponde à subpalavra acima de x . Observe que: (1) a única desigualdade de x_{19} e x_{39} implica mais que uma desigualdade em $\eta(1)$; (2) existe redundância entre os núcleos, como por exemplo, na subpalavra 3 2 6 de $\eta(1)$, o símbolo 2 não é realmente necessário, já que o núcleo 3 intersecciona o núcleo 6 e (3) sempre que necessário pode-se manter o número de símbolos no começo de cada núcleo antes do próximo núcleo começar. Por exemplo, para o núcleo de nome 3 acima este número é 3 e para o núcleo 2 este número é 2.

A razão para utilização do sistema alternativo é que esse sistema permite satisfazer a seguinte propriedade. Considere dois núcleos de algum $\eta(i)$ que representam duas subpalavras idênticas de x . Então, essas subpalavras devem possuir o mesmo nome. Esta propriedade é mais forte que a propriedade da consistência da partição e assegura que o terceiro estágio pode ser implementado eficientemente em paralelo, sem perda de qualquer identidade.

O terceiro estágio constrói a árvore dos sufixos associada à palavra x da seguinte forma. A entrada para a última iteração é a árvore dos sufixos $T(1)$ dos nomes derivados de $\eta(1)$. A última iteração constrói a árvore dos sufixos dos nomes de $\eta(0) = x$. A i -ésima iteração constrói a árvore dos sufixos $T(i)$ dos nomes derivados de $\eta(i)$, a partir da árvore dos sufixos $T(i+1)$.

Como já mencionado, o algoritmo básico tem complexidade de tempo $O(\log^2 n)$ e complexidade de trabalho $O(n \log \log n)$, para um alfabeto cujo tamanho é um polinômio em n . Além disso, uma implementação alternativa tem complexidade de tempo $O(n^\delta)$ e complexidade de trabalho $O(n \log^* n)$, para $0 < \delta \leq 1$.

Algoritmo Ótimo

Considere inicialmente o caso geral, onde a palavra de entrada é obtida de um alfabeto cujo tamanho é limitado por um polinômio em n . O algoritmo básico não possui trabalho linear devido aos seguintes três problemas:

1. No primeiro estágio, particionando uma palavra de tamanho n em blocos, pode ser necessário um número de operações que é proporcional a $n \log^* n$. A atribuição de nomes aos núcleos no segundo estágio também necessita da realização de $O(n \log^* n)$ operações.
2. No terceiro estágio, necessita-se executar a ordenação, em cada iteração, de n inteiros, selecionados em um intervalo de $1, \dots, n^2$. Até o momento, não existe um algoritmo paralelo que soluciona este problema de forma ótima em tempo polilogarítmico.
3. O refinamento de $T(i)$, na iteração i do terceiro estágio, pode necessitar de $\log^* n$ passos para cada vértice, produzindo um número de operações que é proporcional a $n \log^* n$.

O algoritmo ótimo é então desenvolvido supondo que a palavra de entrada x seja obtida de um alfabeto cujo tamanho é limitado por alguma constante c .

O algoritmo ótimo inicia da seguinte forma. As primeiras $O(\log \log^* n)$ iterações do algoritmo básico são modificadas. O objetivo é reduzir o tamanho de $x(i)$ e $\eta(i)$ para menos que $n/(\log^* n)^2$. A saída das iterações de 1 a $O(\log \log^* n)$ no primeiro e segundo estágios do algoritmo básico são computadas utilizando um algoritmo alternativo. Equivalentemente, as últimas $\log \log_c \log n$ iterações do terceiro estágio do algoritmo básico são trocadas por um algoritmo alternativo. As outras iterações dos três estágios do algoritmo básico são mantidas inalteradas. O projeto do algoritmo ótimo é essencialmente uma aplicação da técnica de aceleração em cascata.

O algoritmo ótimo tem complexidade de tempo $O(\log^2 n)$ e complexidade de trabalho $O(n)$, para uma palavra obtida de um alfabeto cujo tamanho é constante.

4.2.4 Algoritmo Hariharan

O **algoritmo Hariharan**, proposto em [Har94], constrói a árvore dos sufixos associada à palavra x , de tamanho n , sobre um alfabeto Σ de tamanho constante, com complexidade de tempo $O(\log^4 n)$, complexidade de trabalho $O(n)$, no modelo CRCW PRAM. O algoritmo utiliza espaço $O(n)$ e foi o primeiro algoritmo paralelo ótimo no trabalho e no espaço utilizado para a solução desse problema. Ele pode ainda ser generalizado para

uma palavra x obtida de um alfabeto geral, polinomial em n , com desempenho de tempo $O(\log^4 n)$ e de trabalho $O(n \log |\Sigma|)$, utilizando espaço $O(n \log |\Sigma|)$. Neste caso também, o algoritmo é de trabalho ótimo.

Note que a construção da árvore dos sufixos da palavra x envolve implicitamente a ordenação dos símbolos de x através do alfabeto Σ . Portanto, nenhum algoritmo paralelo pode ter um limitante melhor para o trabalho que o **algoritmo AILSV** quando os símbolos em x são obtidos de um alfabeto de tamanho polinomial em n . Entretanto, o **algoritmo AILSV** possui as mesmas complexidades de tempo, trabalho e espaço para, por exemplo, alfabetos binários. A existência de um algoritmo de complexidade de trabalho linear, para o caso em que a palavra x é obtida de um alfabeto de tamanho constante, foi um importante desafio no processamento paralelo de palavras, solucionado em [SV94, Har94].

O **algoritmo AILSV** utiliza algumas técnicas algorítmicas interessantes para a construção da árvore dos sufixos. No entanto, nesse algoritmo não existe o uso das propriedades intrínsecas de palavras. A técnica utilizada em [Har94] é bastante diferente e explora propriedades combinatoriais e periódicas de palavras, devido especialmente à natureza repetitiva da árvore dos sufixos. É interessante notar que, nesse algoritmo, é explorada a suposição de que o alfabeto é de tamanho constante para a obtenção de um algoritmo de trabalho ótimo, utilizando espaço linear. A seguir apresentaremos os dois principais ingredientes do algoritmo desenvolvido:

- O esquema geral do algoritmo depende de um limitante superior, que será descrito, para a seguinte questão combinatória. Se i_y é o número de vezes que a subpalavra y ocorre em x , então qual o valor do somatório $\sum \min\{i_y, i_{0y}\}$, sobre todas as palavras binárias y , de tamanho m , como uma função de n e de m ? Note que este somatório está estritamente relacionado com o número de vértices da árvore dos sufixos associada a x , que possuem duas ligações sufixas incidentes sobre eles. Em [Har94] mostrou-se que esse somatório é limitado superiormente por $(n \log n)/m$.
- O algoritmo paralelo está baseado em uma versão concorrente do algoritmo seqüencial de McCreight para construção da árvore dos sufixos, e constrói uma única estrutura de dados na qual as árvores dos sufixos das palavras binárias $x_{\{1\}}, \dots, x_{\{k\}}$ são intercaladas. O algoritmo tem complexidade de tempo $O(\max_i \{|x_{\{i\}}|\} \log k)$, complexidade de trabalho $O(\sum_i |x_{\{i\}}|)$, e utiliza espaço $O(\sum_i |x_{\{i\}}|)$. Neste algoritmo, um processador é atribuído para cada palavra $x_{\{i\}}$; cada processador executa o algoritmo seqüencial de McCreight em sua respectiva palavra. Isto leva a dois problemas, e o mais crítico deles é o fato que um processador que insere um vértice v nesta estrutura de dados pode ter de esperar pela ligação sufixa de seu vértice pai antes de continuar. De fato, longos caminhos podem aparecer na estrutura de dados, com cada vértice deste caminho esperando pela ligação sufixa de seu vértice pai. Um argumento de amortização é dado para mostrar que o trabalho total realizado é de fato linear.

Visão Geral do Algoritmo

O **algoritmo Hariharan** utiliza o seguinte esquema para construção da árvore dos sufixos. Árvores dos sufixos são construídas para subpalavras de x de tamanhos sucessivamente crescentes, em $O(\log n)$ estágios; a árvore final produzida é a árvore dos sufixos associada à palavra x . No primeiro estágio, as subpalavras têm tamanho m , onde m é polilogarítmico em n . Nos estágios subseqüentes, as subpalavras têm tamanhos $2m, 2^2m, 2^3m, \dots$

O primeiro estágio estabelece o primeiro desafio deste problema, já que não é claro nem mesmo como este primeiro estágio pode ser completado com complexidade de trabalho $O(n)$. A versão concorrente do algoritmo de McCreight descrita anteriormente é utilizada da seguinte forma. A palavra x é dividida em blocos de tamanho $2m$. Um processador é atribuído para cada bloco. Cada processador executa o algoritmo seqüencial de McCreight para inserir todas as subpalavras de tamanho m de seu bloco em uma árvore dos sufixos. Todos os processadores inserem subpalavras na mesma estrutura de dados. O tempo utilizado até esse momento é $O(m \log n)$ e o trabalho $O(n)$.

Cada estágio subseqüente pode ser executado facilmente pela ordenação dos n itens em cada estágio. Entretanto, o trabalho realizado na execução dessa tarefa é $O(n \log^2 n)$. O principal desafio nesse instante é encontrar uma alternativa para o problema de ordenar n itens em cada estágio. Para evitar que a complexidade de trabalho seja ineficiente como acima, o seguinte esquema é utilizado.

Cada estágio é executado em dois passos. Como visto anteriormente, em um dado estágio todas as subpalavras de um determinado tamanho são processadas. No primeiro passo, a árvore dos sufixos é computada para um subconjunto destas subpalavras cuidadosamente escolhido. Utilizando a propriedade combinatória mencionada acima e explorando propriedades periódicas, pode-se mostrar que o número de tais subpalavras que não são periódicas com períodos pequenos é $O(n/\log^2 n)$, e que aquelas subpalavras que são periódicas com pequenos períodos podem ser agrupadas em $O(n/\log^2 n)$ famílias. Pode-se então ordenar estas subpalavras escolhidas utilizando o algoritmo paralelo de ordenação por intercalação de Cole [Col88] de tal forma que o trabalho total realizado na ordenação é $O(n/\log n)$ por estágio e, portanto, $O(n)$ em todos os estágios.

No segundo passo, explorando a estrutura repetitiva da árvore dos sufixos, obtém-se a árvore dos sufixos associada a todas as subpalavras utilizando a árvore dos sufixos associada às subpalavras escolhidas acima. Estas subpalavras escolhidas permitem que um grafo definido sobre as folhas da árvore dos sufixos obtida no estágio anterior seja particionado em árvores; este particionamento do grafo é crítico para o desempenho eficiente do segundo passo.

Finalmente, o principal resultado obtido em [Har94] é apresentado no Teorema 4.2.20 a seguir.

Teorema 4.2.20 *Existe um algoritmo paralelo no modelo PRAM CRCW comum para construir a árvore dos sufixos associada a uma palavra x de tamanho n , sobre um alfabeto de tamanho constante, com complexidade de tempo $O(\log^4 n)$, complexidade de trabalho $O(n)$ e que utiliza espaço $O(n)$. Além disso, existe um algoritmo paralelo no modelo PRAM CRCW comum para construir a árvore dos sufixos associada a uma palavra x de tamanho n , sobre um alfabeto Σ de tamanho polinomial em n , com complexidade de tempo $O(\log^4 n)$, complexidade de trabalho $O(n \log |\Sigma|)$ e que utiliza espaço $O(n \log |\Sigma|)$, sendo ordenados os símbolos em Σ . Ambos os algoritmos podem ser implementados no modelo PRAM CREW com as mesmas complexidades.*

□

Capítulo 5

Aplicações

A utilidade da árvore dos sufixos pode ser ilustrada em vários problemas de processamento de palavras e em alguns problemas de outras áreas da ciência da computação. As soluções paralelas de tais problemas, que constroem esta estrutura em paralelo no primeiro estágio de pré-processamento, apresentam níveis de dificuldade diferentes. Ou seja, existem tanto aplicações bastante simples e imediatas como aplicações sofisticadas e bem mais elaboradas.

Estamos, na verdade, interessados na utilização da árvore dos sufixos no projeto de algoritmos paralelos eficientes para solução de diversos problemas em ciência da computação, especialmente problemas de processamento de palavras. Dessa forma, para cada problema apresentado, descrevemos uma solução paralela, possivelmente indicando referências que o solucionam seqüencialmente. Além disso, as soluções são apresentadas com diferentes níveis de profundidade.

Os problemas de encontrar a maior subpalavra repetida de uma dada palavra e de encontrar o identificador de uma subpalavra, dados um índice e uma palavra, são aplicações imediatas da árvore dos sufixos e, portanto, vêm descritos inicialmente. O problema do casamento de padrões *on-line* (*on-line string matching*) também pode ser solucionado com a utilização da árvore dos sufixos e, devido a sua importância, descrevemos cuidadosamente tal aplicação. Em seguida, aplicações mais sofisticadas são descritas, com menos profundidade: encontrar o menor ancestral comum, casamento aproximado de padrões (*approximate string matching*), construção do autômato dos sufixos minimal e do autômato dos fatores minimal, a generalização da árvore dos sufixos associada a matrizes quadradas e compressão de dados. Alguns desses problemas possuem aplicações em outras áreas, tais como o casamento aproximado de padrões e a generalização da árvore dos sufixos para matrizes quadradas, com aplicações em biologia molecular computacional e em processamento digital de imagens de baixo nível, respectivamente.

5.1 Maior Subpalavra Repetida

Descrição do problema

Dada uma palavra x encontrar a maior subpalavra que ocorre mais que uma vez em x .

Solução ([JáJ92])

Seja x uma palavra de tamanho n e suponha, sem perda de generalidade, que $n = 2^k$, para algum inteiro positivo k , e que $x_n = \#$ é um símbolo especial que não ocorre em nenhuma outra posição de x . O pré-processamento consiste de construir a árvore dos sufixos T_x associada a x e computar $|\mathcal{W}(v)|$ para cada vértice interno $v \in V \setminus F$ de T_x . A computação de $|\mathcal{W}(v)|$, para todo $v \in V \setminus F$, é armazenada em um novo atributo da matriz $Vért$, e pode ser realizada de duas maneiras: durante a execução do algoritmo **AILS**V, atualizando o atributo $|\mathcal{W}(v)|$ sempre que necessário, ou logo após a construção da árvore dos sufixos T_x , utilizando uma técnica básica em computação paralela, chamada *pointer jumping* [JáJ92]. Assim, o pré-processamento é executado em tempo paralelo $O(\log n)$, utilizando $O(n \log n)$ operações.

A resposta para uma consulta **maior-subpalavra-repetida** é um vértice interno $u \in V \setminus F$ de T_x tal que $|\mathcal{W}(u)| = \max_{v \in V \setminus F} |\mathcal{W}(v)|$.

Pela Proposição 3.3.3, para uma árvore dos sufixos T_x associada a uma palavra x de tamanho n , temos que $|V| \leq 2n - 1$. Logo, $|V \setminus F| \leq n - 1$. Portanto, nosso problema se reduz a encontrar o máximo de $O(n)$ elementos. Tal problema pode ser solucionado utilizando a técnica de aceleração em cascata (*accelerated cascading*) [JáJ92] em tempo paralelo $O(\log \log n)$, utilizando um número linear de operações, no modelo PRAM CRCW comum. Outra alternativa de solução para o problema de encontrar o máximo de uma sequência de n elementos em paralelo está descrita em [SV81], e apresenta as mesmas complexidades citadas.

Portanto, o processamento da consulta **maior-subpalavra-repetida** pode ser executado em tempo paralelo $O(\log \log n)$, utilizando um número linear de operações. Um exemplo deste tipo de consulta é ilustrado na Figura 5.1.

5.2 Identificadores de Subpalavras

Descrição do problema

Dado i , encontrar a menor subpalavra w de uma palavra x que identifica a posição i de x ; ou seja, w ocorre em x na posição i , mas w não ocorre em nenhuma outra posição de x .

Entrada: $x = abbabba\#$

Maior subpalavra repetida: $\mathcal{W}(v_5) = abba$.

Figura 5.1: Um exemplo de consulta **maior-subpalavra-repetida**.

Solução ([JáJ92])

Supomos, sem perda de generalidade, que x é uma palavra de tamanho $n = 2^k$ tal que $x_n = \#$ é um símbolo especial que não ocorre em nenhuma outra posição de x . Neste caso, o sufixo $x_i x_{i+1} \dots x_n$ é um identificador da posição i de x , mas um prefixo menor pode ser suficiente.

Na fase de pré-processamento, basta construir a árvore dos sufixos T_x , associada à palavra x , conforme o **algoritmo AILSV** apresentado na Seção 4.2.2.

Dado i , com $1 \leq i \leq n$, podemos localizar a folha $v \in F$ de T_x , associada ao sufixo $x_i x_{i+1} \dots x_n$ na matriz $Vért$, em tempo seqüencial constante. A resposta a essa consulta é a concatenação da palavra $\mathcal{W}(\mathcal{P}(v))$ com o primeiro símbolo do rótulo da aresta $\mathcal{P}(v) \rightarrow v$ de T_x .

O algoritmo de pré-processamento pode ser implementado em tempo paralelo $O(\log n)$, utilizando um total de $O(n \log n)$ operações, e a consulta **identificador-de-subpalavra** pode ser respondida em tempo seqüencial $O(1)$. Um exemplo de uma consulta deste tipo é ilustrado na Figura 5.2.

Entrada: $x = abbabba\#$ e $i = 3$

Identificador da subpalavra que inicia na posição 3 de x : *bab*

Figura 5.2: Um exemplo de consulta **identificador-de-subpalavra**.

5.3 Casamento de Padrões *On-line*

Descrição do problema

Dada uma palavra x , chamada **texto**, de tamanho n , para pré-processamento, com $n = 2^k$ e $x_n = \#$ um símbolo especial que não ocorre em nenhuma outra posição de x , responder rapidamente a consultas da forma: a palavra y , chamada **padrão**, de tamanho m , com $m \leq n$, ocorre em x ?

Como já mencionado, a árvore dos sufixos foi inicialmente concebida com o objetivo de solucionar esse problema. Diversos algoritmos seqüenciais tratam especificamente desta aplicação da árvore dos sufixos [Wei73, Aku93, Kos94].

Vejam agora a solução paralela proposta por [AIL⁺88]. Claramente, qualquer consulta envolvendo o padrão y requer $\Omega(m)$ operações, já que y não está disponível para pré-processamento, e portanto todos os símbolos de y devem ser examinados. O objetivo então é executar tal consulta em tempo $O(\log m)$, utilizando $O(m)$ operações, independentemente do tamanho da palavra x . Veremos a seguir que este objetivo é de fato possível com a utilização da árvore dos sufixos. Antes do início desta descrição, a seguinte definição se faz necessária.

Definição 5.3.1 *Seja x uma palavra e y uma subpalavra de x . O locus $\mathcal{L}(y)$ de y na árvore dos sufixos T_x é um vértice $v \in V$ de T_x tal que y é prefixo de $\mathcal{W}(v)$ e $\mathcal{W}(\mathcal{P}(v))$ é um prefixo próprio de y .*

Por exemplo, na Figura 5.1, o locus da palavra abb é o vértice v_5 , isto é, $\mathcal{L}(abb) = v_5$.

Pela Definição 5.3.1 acima, uma palavra qualquer y ocorre em x se e somente se existe um vértice $v \in V$ de T_x tal que $\mathcal{L}(y) = v$. A resposta a uma consulta da forma “ y ocorre em x ?” é um vértice deste tipo.

Solução (por [AIL⁺88])

Como x está disponível para pré-processamento, construímos a árvore dos sufixos T_x associada à palavra x . Durante o processo de construção de T_x , as seguintes informações devem ser guardadas:

1. Todas as $\log n$ matrizes BB utilizadas no procedimento *Árvore Esqueleto*;
2. Todas as árvores intermediárias $D^{(k)}$, com $k = \log n, \dots, 0$. Aos vértices internos de cada árvore intermediária $D^{(k)}$ devem estar associados seus respectivos vetores de ramificações R .

Agora, para cada vértice interno $u \in V$ de $D^{(k)}$, com $0 \leq k \leq \log n - 1$, é atribuído um vetor de ramificações de tamanho n , chamado R_u . O vetor R_u armazena as ramificações, ou as arestas descendentes de u , da seguinte forma. Se o vértice $v \in V$ de $D^{(k)}$ é tal que $\mathcal{P}(v) = u$ e o *Rótulo* ascendente contido em v é (i, l) , então um apontador para v é armazenado em $R_u[DS[i, k]]$, sempre que exista uma subpalavra de x de tamanho pelo menos 2^k , que rotula a aresta $v \rightarrow u$, ou seja, sempre que $2^k \geq l$.

Um exemplo completo da fase de pré-processamento, incluindo todas as informações que necessitam ser armazenadas, conforme os itens (1) e (2) acima, é ilustrado nas Figuras 5.3, 5.4 e 5.5. Neste exemplo, a palavra $x = abbabba\#$ é fornecida como entrada.

O processamento de consultas **casamento-de-padrões** *on-line* será descrito algoritmicamente a seguir, em um nível bastante alto, no procedimento *Casamento de Padrões*. Basicamente, este procedimento consiste de três passos: no Passo 1, construímos a matriz DS_y dos descritores de subpalavras especiais de y ; no Passo 2, um teste inicial é realizado para que, no Passo 3, iterações sejam realizadas com o objetivo de encontrar um vértice v em T_x tal que $\mathcal{L}(y) = v$, se tal vértice existe. Sempre que necessário, utilizaremos a notação BB_i , com $i = 0, \dots, \log n - 1$, para referenciar a matriz BB na i -ésima iteração do procedimento *Árvore Esqueleto*. Da mesma forma, utilizaremos a notação $Vért_i$ e R_u^i , com $i = \log n - 1, \dots, 0$, para referenciar a matriz $Vért$ e os vetores R_u , com $u \in V$ de $D^{(i)}$, no i -ésimo refinamento sucessivo de $m-D_x$.

Figura 5.3: Algumas estruturas salvas na fase de pré-processamento: a matriz dos descritores DS_x da palavra $x = abbabba\#$ e as matrizes BB das iterações do procedimento *Árvore Esqueleto*.

$D^{(2)}$

	0	1	2	3
1	(1,1)		(2,3)	(5,4)
	v_0		(1,0)	(1,2)
2	(2,1)	(3,1)		(4,6)
	v_0	(2,0)		(2,1)
3		(4,1)		(5,6)
		(2,0)		(3,1)
4				(8,4)
				(1,2)
5				(7,6)
				(2,1)
6				(8,6)
				(3,1)
7				(8,7)
				(1,0)
8				(8,8)
				v_0

	1	2	3	4	5	6	7	8
v_0	(1,0)	(2,0)						(8,3)
(1,0)		(1,2)						(7,3)
(2,0)	(3,1)	(2,1)						
(2,1)	(2,3)						(5,3)	
(3,1)					(3,3)			(6,3)
(1,2)					(1,3)			(4,3)

$D^{(1)}$

	0	1	2	3
1	(1,1)		(2,3)	(5,4)
	v_0		(1,0)	(1,2)
2	(2,1)	(3,1)		(4,6)
	v_0	(2,0)		(2,1)
3		(4,1)		(5,6)
		(2,0)		(3,1)
4				(8,4)
				(1,2)
5				(7,6)
				(2,1)
6				(8,6)
				(3,1)
7				(8,7)
				(1,0)
8				(8,8)
				v_0

	1	2	3	4	5	6	7	8
v_0	(1,0)	(2,0)						(8,3)
(1,0)		(1,2)						(7,3)
(2,0)	(3,1)	(2,1)						
(2,1)	(2,3)						(5,3)	
(3,1)		(3,3)						(6,3)
(1,2)		(1,3)						(4,3)

Figura 5.4: As árvores intermediárias $D^{(2)}$ e $D^{(1)}$, juntamente com os vetores de ramificações de seus vértices internos.

	0	1	2	3		1	2	3	4	5	6	7	8
1	(1,1)		(2,3)	(5,4)	v_0	(1,0)	(2,0)						(8,3)
2	(2,1)		(3,2)	(5,5)	v_0		(1,2)						(7,3)
3		(4,1)		(5,6)			(2,0)	(2,2)					
4				(8,4)									
5				(1,2)									
6				(8,5)									
7				(2,2)									
8				(8,6)									
				(3,1)									
				(8,7)									
				(1,0)									
				(8,8)									
				v_0

Figura 5.5: A árvore $D^{(0)}$, juntamente com os vetores de ramificações de seus vértices internos, e a árvore dos sufixos T_x associada à palavra $x = abbabba\#$.

Procedimento Casamento de Padrões

Entrada: as $\log n$ matrizes BB geradas no procedimento *Árvore Esqueleto*, as $\log n$ matrizes $Vért$, geradas nos refinamentos sucessivos de $m-D_x$, representando as árvores intermediárias $D^{(k)}$, com $k = \log n - 1, \dots, 0$, juntamente com os vetores de ramificações dos vértices internos de cada árvore intermediária; e o padrão y de tamanho m , tal que $m \leq n$.

Saída: uma resposta “SIM” indicando que o padrão ocorre no texto x ou uma resposta “NÃO”, caso contrário; e um vértice $v \in V$ de $D^{(k)}$, com $0 \leq k \leq \log n$, indicando respectivamente que $\mathcal{L}(y) = v$ ou que y é um prefixo próprio de $\mathcal{W}(v)$.

Passo 1. No procedimento *Árvore Esqueleto* computamos os nomes das subpalavras de $x\#$ de tamanho potência de 2 de tal forma que o valor $DS_x[i, q]$, com $1 \leq i \leq n$ e $0 \leq q \leq \log n$, é um nome único dado à subpalavra $x_i x_{i+1} \dots x_{i+2^q-1}$ de $x\#$ e $DS_x[i, q] = DS_x[j, q]$ se e somente se $x_i x_{i+1} \dots x_{i+2^q-1} = x_j x_{j+1} \dots x_{j+2^q-1}$. Neste passo, realizamos a atribuição de nomes a algumas subpalavras do padrão y . Para cada $q = 0, \dots, \lfloor \log m \rfloor$, as subpalavras que estarão recebendo nomes são aquelas que possuem tamanho 2^q e começam na posição i , onde $i - 1$ é um múltiplo de 2^q e $i + 2^q \leq m + 1$. Os nomes são armazenados em uma matriz DS_y dos descritores das subpalavras da palavra padrão y , de dimensão $m \times \lfloor \log m \rfloor + 1$, equivalentemente à matriz DS_x . A atribuição de nomes é realizada de forma que se duas subpalavras de tamanho 2^q , uma em $x\#$ e outra em y , são idênticas então seus nomes são iguais. Essa computação pode ser feita utilizando as matrizes BB geradas no procedimento *Árvore Esqueleto* do algoritmo AILSV.

Passo 2. Seja $DS_y[1, \lfloor \log m \rfloor] = k$ o nome do prefixo de y de tamanho $2^{\lfloor \log m \rfloor}$. Observe que se nenhum $DS_x[i, \lfloor \log m \rfloor]$ é igual a k então o prefixo de y cujo tamanho é $2^{\lfloor \log m \rfloor}$ não ocorre em x . Podemos então concluir que a resposta a essa consulta é “NÃO”, em conjunto com o vértice raiz v_0 , isto é, y não ocorre em x e $\mathcal{L}(y) = v_0$ em T_x . Suponha que $DS_x[i, \lfloor \log m \rfloor] = k$, para algum i , onde $1 \leq i \leq n - 1$. Devemos verificar agora se $Vért[k, \lfloor \log m \rfloor]$ aparece em $D^{(\log n - 1)}$. Note que, pela Definição 4.2.2 da árvore esqueleto D_x , $Vért[k, \lfloor \log m \rfloor]$ não aparecerá em $D^{(\log n - 1)}$ se e somente se todas as subpalavras de $x\#$ cujo prefixo de tamanho $2^{\lfloor \log m \rfloor}$ é idêntico ao prefixo de y de mesmo tamanho, também possuem o mesmo prefixo de tamanho $2^{\lfloor \log m \rfloor + 1}$. Se $Vért[k, \lfloor \log m \rfloor]$ aparece em $D^{(\log n - 1)}$ então, como os refinamentos $D^{(\log n - 1)} \rightarrow D^{(\log n - 2)} \rightarrow \dots \rightarrow D^{(\lfloor \log m \rfloor)}$ manipulam somente subpalavras de tamanho maior que $2^{\lfloor \log m \rfloor}$, o vértice $Vért[k, \lfloor \log m \rfloor]$ também aparecerá em $D^{(\lfloor \log m \rfloor)}$ e prosseguimos com o Passo 3. Caso contrário, isto é, se $Vért[k, \lfloor \log m \rfloor] = \lambda$ em $D^{(\log n - 1)}$, então devemos verificar se y é igual a $x_k x_{k+1} \dots x_{k+m-1}$ símbolo a símbolo.

Passo 3. O objetivo desse passo é encontrar um vértice v na árvore dos sufixos T_x tal que y é um prefixo de $\mathcal{W}(v)$, se tal vértice existe. Para isto, utilizamos a matriz DS_y computada no Passo 1 e as árvores $D^{(q)}$, $q = \lfloor \log m \rfloor - 1, \dots, 0$, do pré-processamento. O vértice v é encontrado usando alguma noção de busca binária em $\lfloor \log m \rfloor$ iterações, descritas a seguir.

Iteração q , ($q = \lfloor \log m \rfloor - 1, \dots, 0$). Sejam v e y' os parâmetros de entrada da iteração q . Para a iteração $\lfloor \log m \rfloor - 1$, $v = \text{Vért}_{\lfloor \log m \rfloor}[k, \lfloor \log m \rfloor]$ e y' é o sufixo de y que começa na posição $2^{\lfloor \log m \rfloor + 1}$. O objetivo então é verificar se y' ocorre em $\mathcal{W}(v)$. A árvore que estamos trabalhando é a árvore $D^{(q)}$. Existem duas possibilidades:

1. O vértice v aparece em $D^{(q)}$. A Possibilidade 1 possui duas subpossibilidades:
 - (a) 2^q é maior que o tamanho de y' . Neste caso, não há nada a fazer e os parâmetros de entrada da iteração atual tornam-se os parâmetros de entrada da próxima iteração.
 - (b) 2^q é menor ou igual ao tamanho de y' . Suponha que y' começa na posição j de y e γ é o valor armazenado em $DS_y[j, q]$. Se o valor $R_v[\gamma]$ é vazio então y não ocorre em x . Caso contrário, os parâmetros de entrada da próxima iteração serão o sufixo de y' que começa na posição $2^q + 1$ e o vértice apontado por $R_v[\gamma]$.
2. O vértice v não aparece em $D^{(q)}$. Isto significa que v tem somente um filho em $\hat{D}^{(q+1)}$ e assim v foi omitido de $D^{(q)}$ no Passo 2 do refinamento $D^{(q+1)}$. Seja u o único filho de v em $\hat{D}^{(q+1)}$. A Possibilidade 2 possui duas subpossibilidades:
 - (a) 2^q é maior que o tamanho de y' . Suponha que o Rótulo de u em $D^{(q)}$ é (i, l) . Neste caso, y' ocorre em x se e somente se y' é um prefixo de $x_{i+l-2^q+1}, \dots, x_{i+l}$. Verificamos isto símbolo a símbolo.
 - (b) 2^q é menor ou igual ao tamanho de y' . Comparamos $DS_x[i+l-2^q+1, q]$, o único nome de $x_{i+l-2^q+1}, \dots, x_{i+l}$, com o único nome do prefixo de y' cujo tamanho é 2^q . Se estes nomes são diferentes então y não ocorre em x . Caso contrário, os parâmetros de entrada da próxima iteração serão o sufixo de y' começando na posição $2^q + 1$ e o vértice u .

A computação da matriz DS_y dos descritores de algumas subpalavras de y é equivalente a construção da matriz DS_x dos descritores dos fatores básicos de x , realizada corretamente no procedimento *Árvore Esqueleto*, na Seção 4.2.2, conforme o Lema 4.2.3. Dessa forma, podemos afirmar que o Passo 1 do procedimento *Casamento de Padrões* computa corretamente a matriz DS_y em tempo paralelo $O(\log m)$. No entanto, o número total de subpalavras de y envolvidas nesse processo é $\leq \sum_{q=0}^{\lfloor \log m \rfloor} \frac{m}{2^q} = O(m)$, e portanto, o número total de operações realizadas nesse passo é $O(m)$.

O Passo 2 verifica se existe alguma subpalavra de x , de tamanho $\lfloor \log m \rfloor$, idêntica ou com o mesmo nome do prefixo de tamanho $\lfloor \log m \rfloor$ de y . Se tal subpalavra não existe, então certamente o padrão y não ocorre em x e a resposta para a consulta **casamento-de-padrões** é “NÃO”. Caso contrário, é verificado se existe um vértice $v \in V$ de $m\text{-}D_x = D^{(\log n-1)}$, de alguma forma relacionado a essa subpalavra. Se esse vértice $v \in V$ existe em $D^{(\log n-1)}$, podemos garantir que v existe em $D^{(\lfloor \log m \rfloor)}$, já que os refinamentos $D^{(\log n-1)} \rightarrow$

$\dots \rightarrow D^{(\lfloor \log m \rfloor)}$ processam somente subpalavras de tamanho maior que $2^{\lfloor \log m \rfloor}$ e então segue-se ao Passo 3. Caso contrário, isto é, se o vértice v não existe em $D^{(\log n - 1)}$, deve-se verificar se y é igual a $x_k x_{k+1} \dots x_{k+m-1}$ símbolo a símbolo. Essa tarefa pode ser executada em tempo $O(\log m)$, utilizando $O(m)$ operações. Claramente, essas complexidades são as complexidades do Passo 2.

A busca de um vértice v na árvore dos sufixos T_x tal que y é um prefixo de $\mathcal{W}(v)$, se tal vértice existe, é realizada durante a execução do Passo 3. O vértice v é encontrado utilizando-se alguma noção de busca binária em $\lfloor \log m \rfloor$ iterações, com o uso da matriz DS_y do Passo 1 e as árvores intermediárias $D^{(q)}$, com $q = \lfloor \log m \rfloor - 1, \dots, 0$, da fase de pré-processamento. O vértice v e a subpalavra y' são os parâmetros de entrada da q -ésima iteração, com $q = \lfloor \log m \rfloor - 1, \dots, 0$. Para a iteração $q = \lfloor \log m \rfloor - 1$, v é o vértice ao final do Passo 2 e y' é o sufixo de y que começa na posição $2^{\lfloor \log m \rfloor + 1}$. O objetivo neste instante é verificar se y' ocorre em $\mathcal{W}(v)$, trabalhando em $D^{(q)}$. A propriedade invariante satisfeita em todas as iterações é que $v \in V$ de $D^{(q+1)}$ e y' é uma subpalavra cujo tamanho é menor que 2^{q+1} . A corretude desse passo pode ser demonstrada por indução em q , utilizando a propriedade invariante e levando em consideração as possibilidades e subpossibilidades de cada iteração. O tempo gasto nesse passo também é $O(\log m)$ e o número total de operações utilizadas é $O(m)$. Essa discussão que acabamos de apresentar motiva a apresentação do seguinte teorema:

Teorema 5.3.2 *Dadas uma palavra texto x e uma palavra padrão y , de tamanhos n e m , respectivamente, com $m \leq n$, a resposta a uma consulta do tipo **casamento-de-padrões** é computada corretamente pelo procedimento Casamento de Padrões em tempo paralelo $O(\log m)$, realizando trabalho $O(m)$, em uma PRAM CRCW.*

□

Um exemplo foi selecionado para ilustrar a execução do procedimento *Casamento de Padrões*, apresentado na Figura 5.6.

5.4 Outras Aplicações

Nesta seção estamos interessados apenas em comprovar a ampla utilização da árvore dos sufixos na solução de diversos problemas de algumas áreas da ciência da computação. Dessa forma, apresentaremos sucintamente certas aplicações, apenas descrevendo o problema e relacionando referências que propõem algoritmos, seqüenciais e paralelos, que solucionam tal problema utilizando a árvore dos sufixos como um estágio inicial de pré-processamento.

Entrada: $x = abbabba\#$ e $y = abbabba$

Passo 1	Passo 2	Passo 3
<div><div><div><div>012</div><div>111</div><div>22</div><div>323</div><div>41</div><div>522</div><div>62</div><div>71</div></div><div><div><div><div>DS_y</div><div>1</div><div>2</div><div>3</div><div>4</div><div>5</div><div>6</div><div>7</div></div><div><div><div>1</div><div>2</div><div>3</div><div>4</div><div>5</div><div>6</div><div>7</div></div></div></div></div></div><div>$DS_y[1, 2] = 1 = DS_x[1, 2]$ $Vért[1, 2] \in D^{(2)}$</div></div>	<div>$q = 1, v = Vért[1, 2]$ e $y' = bba$ 1. $v \in V$ de $D^{(1)}$ 1.2. $2^q \leq y'$ $DS_y[5, 1] = 2, R_v[2] = (1, 3)$ $q = 0, v = Vért[1, 3]$ e $y' = a$ 1. $v \in F$ de $D^{(0)}$ e $y' \neq \epsilon$ $x_m = y'$</div>	
Resposta "SIM", e devolva $v = Vért[1, 3] \in V$ de $D^{(0)}$.		

Figura 5.6: O procedimento *Casamento de Padrões* sendo executado, dadas a palavra texto $x = abbabba\#$ e a palavra padrão $y = abbabba$.

Menor Ancestral Comum

Descrição do problema

Seja T uma árvore com raiz, disponível para pré-processamento. Responder a consultas em tempo real da forma: "qual vértice é o menor ancestral comum (*lowest common ancestor*) de u e v ?", para qualquer par de vértices $u, v \in V$ de T .

Solução seqüencial

O algoritmo proposto por [HT84] requer tempo $O(n)$ para pré-processamento e tempo $O(1)$ por consulta.

Solução paralela

O algoritmo desenvolvido em [SV88] considera que a árvore dos sufixos T_x , associada à palavra x de tamanho n , está disponível para pré-processamento. Nesta fase, a árvore dos sufixos T_x sofre modificações em $O(\log n)$ passos paralelos, utilizando $O(n)$ operações. Essa nova estrutura suporta eficientemente consultas do tipo **menor-ancestral-comum**: k consultas podem ser processadas em tempo $O(1)$, utilizando $O(k)$ operações, no modelo PRAM EREW. O algoritmo proposto por [LO92] soluciona o problema de computar o

menor ancestral comum de todos os pares de vértices de uma árvore. Para a árvore dos sufixos T_x , associada à palavra x de tamanho n , com $|V| = O(n)$, esse algoritmo tem complexidade de tempo $O(\log n)$, utilizando $O(n^2)$ operações, no modelo PRAM EREW. Pela Proposição 3.3.3 é fácil ver que existem $O(n^2)$ pares de vértices diferentes em T_x , e assim, esse algoritmo é ótimo no sentido fraco.

Finalmente, a consulta **menor-ancestral-comum** pode ser facilmente modificada para responder a consultas do tipo: “dadas duas subpalavras y_1 e y_2 de uma palavra x , qual é o menor prefixo comum entre y_1 e y_2 ?”.

Casamento Aproximado de Padrões

Descrição do problema

Sejam $x = x_1, \dots, x_n$ uma palavra texto, $y = y_1, \dots, y_m$ uma palavra padrão, com $m \leq n$, e um inteiro $k \geq 1$. No conhecido problema de casamento de padrões buscam-se todas as ocorrências do padrão no texto. No problema do **casamento aproximado de padrões** (*approximate string matching*) buscam-se todas as ocorrências do padrão no texto, com a existência de no máximo k diferenças. Esse problema também é conhecido como **casamento de padrões com k diferenças** (*string matching with k differences*).

Por exemplo, sejam $abcdefghi$ o texto, $b\alpha d\beta egh$ o padrão e $k = 3$. Na verificação da ocorrência do padrão, com $\leq k = 3$ diferenças, na posição 2 do texto, a seguinte correspondência entre $bcdefgh$ e $b\alpha d\beta egh$ pode ser estabelecida:

1. b (do texto) corresponde a b (do padrão).
2. c a α .
3. d a d .
4. $nada$ a β .
5. e a e .
6. f a $nada$.
7. g a g .
8. h a h .

A correspondência pode ser ilustrada como a seguir:

b	α	d	β	e	g	h
b	c	d		e	f	g

Em somente três lugares existe correspondência entre símbolos diferentes. Isto implica que existe uma ocorrência do padrão, que inicia na posição 2 do texto, com 3 diferenças.

Existem três tipos de diferenças:

1. Um símbolo do padrão corresponde a um símbolo diferente do texto. Neste caso, dizemos que os símbolos **não correspondem** (*mismatch*).
2. Um símbolo no padrão corresponde a um “símbolo vazio” no texto.
3. Um símbolo no texto corresponde a um “símbolo vazio” no padrão.

Consideremos o problema do casamento aproximado de padrões ou casamento de padrões com k diferenças: encontrar todas as ocorrências do padrão no texto com no máximo k diferenças do tipo 1, 2 e 3.

O caso $k = 0$ do problema do casamento aproximado de padrões resume-se no problema do casamento de padrões já estudado por nós na Seção 5.3.

O casamento aproximado de padrões é importante não somente do ponto de vista teórico, mas também do ponto de vista prático. Em particular, o casamento aproximado de padrões é importante em biologia molecular, já que o casamento exato de padrões não é suficiente [CL90, CL94].

Diversas variantes desse problema são consideradas. Por exemplo, quando somente diferenças do tipo (1) são consideradas, esse problema é conhecido como **casamento de padrões com k não correspondências** (*mismatches*). Outra variante é o problema do **casamento aproximado de padrões com trechos de tamanho variável e sem importância** (*approximate string matching with variable-length and don't cares — VLDCs*). Por exemplo, no casamento do padrão *com*tador* com o texto *computador*, o símbolo *** seria substituído pela palavra *put* no texto, com zero diferenças. Por outro lado, casando *com*tador* com *contador*, o número de diferenças seria igual a 1. Tal casamento é particularmente útil quando parte do padrão permite muitas variações.

Apesar de todas estas variantes serem consideradas, o problema do casamento de padrões com k diferenças é o mais importante de todos e algoritmos para solucioná-lo, com a utilização da árvore dos sufixos, serão considerados a seguir.

Soluções seqüenciais

Os algoritmos seqüenciais utilizam o algoritmo em [Wei73] ou em [McC76] para construir a árvore dos sufixos T_x , associada à palavra x , na fase de pré-processamento. Além disso a árvore dos sufixos é modificada pelo algoritmo em [HT84] para suportar a consulta **menor-ancestral-comum**. A fase de pré-processamento tem então complexidade de tempo $O((n + m) \log |\Sigma|)$ e utiliza espaço $O(n + m)$.

O algoritmo de [GG88] soluciona o problema do casamento de padrões com k diferenças com complexidade de tempo $O(nk + (n + m) \log |\Sigma|)$. Para um alfabeto Σ de tamanho constante, o algoritmo de [LV89] tem complexidade de tempo $O(nk)$ e para um alfabeto qualquer a complexidade de tempo é $O(n(\log m + k))$. O melhor algoritmo que conhecemos, proposto por [Ukk93], soluciona este problema com complexidade de tempo $O(m^2q + S)$, onde $q = O(\min(n, m^{k+1}|\Sigma|^k))$ e S é o tamanho da saída, com a utilização da técnica de programação dinâmica sobre a árvore dos sufixos. Finalmente, em [CL90, CL94], o algoritmo desenvolvido tem complexidade de tempo de caso médio $O((n/m)k \log m)$, para $k < m/(\log m + O(1))$. No pior caso, a complexidade é equivalente à dos algoritmos anteriores. Esse algoritmo soluciona problemas mais específicos, onde o texto é aleatório e diferenças não são muito frequentes. Estas restrições são encontradas em diversas tarefas em análise de seqüências de DNA e são bastante caras de se executar com a utilização dos algoritmos anteriores. Não tivemos acesso ao seguinte trabalho recente:

- Cobbs, *Fast Approximate Matching using Suffix Trees*, In Proceedings of 6th Annual Symposium on Combinatorial Pattern Matching, CPM 95.

Soluções paralelas

A fase de pré-processamento utiliza o algoritmo em [AIL⁺88] para construção da árvore dos sufixos T_x associada à palavra x , e também o algoritmo em [SV88] para realizar modificações em T_x de modo que ela suporte a consulta **menor-ancestral-comum**. A fase de pré-processamento tem complexidade de tempo $O(\log m)$, complexidade de trabalho $O((n + m) \log n)$ e utiliza espaço $O(n \log n + m^2)$.

Os algoritmos paralelos descritos em [GG88] solucionam somente o problema do casamento de padrões com k não correspondências. O algoritmo proposto em [LV89] soluciona o problema do casamento de padrões com k diferenças com complexidade de tempo $O(\log m + k)$ e utiliza $O(n \log m + nk)$ operações, também utilizando o modelo PRAM CRCW.

Autômato dos Sufixos Minimal e Autômato dos Fatores Minimal

Descrição do problema

Árvores dos sufixos e grafos orientados acíclicos de palavras (*directed acyclic word graphs* - DAWGs) são estruturas de dados que mantêm relações intrínsecas. Ambas as estruturas são obtidas de uma árvore *trie*, como no exemplo ilustrado na Figura 5.7. Existe uma simetria funcional entre estas estruturas e elas possuem essencialmente a mesma faixa de aplicações.

Em [CR90] foi proposto um algoritmo paralelo que transforma a árvore dos sufixos T_x , associada à palavra x de tamanho n , em um DAWG G_x associado a x , em tempo paralelo $O(\log n)$, utilizando $O(n \log n)$ operações.

Solução paralela (por [CR90])

O autômato dos sufixos minimal e o autômato dos fatores minimal são duas estruturas que podem ser obtidas eficientemente em paralelo, a partir do grafo DAWG.

A computação do autômato dos sufixos minimal é simples: essencialmente o autômato dos sufixos minimal é obtido diretamente do DAWG, mais informações adicionais sobre os estados finais. O algoritmo que implementa essa idéia tem complexidade de tempo paralelo $O(\log n)$, realizando $O(n \log n)$ operações, no modelo PRAM CRCW.

Muito mais complicada é a computação do autômato dos fatores minimal. Um algoritmo com as mesmas complexidades que o que acabamos de mencionar também foi proposto em [CR90].

Árvore dos Sufixos de uma Matriz Quadrada

Descrição do problema

Uma nova estrutura é proposta em [Gia93, Gia95] que generaliza a árvore dos sufixos de uma palavra para a árvore dos sufixos de uma matriz quadrada. Baseados nessa nova estrutura, algoritmos seqüenciais eficientes são propostos para solução de problemas de processamento de imagens de baixo nível e de bancos de dados visuais.

Solução

Somente os algoritmos seqüenciais de [Gia93, Gia95] se propõem a solucionar tal problema. Sua complexidade de tempo é $O(n^2(\log |\Sigma| + \log n))$ e o espaço gasto é $O(n^2)$, para entradas de tamanho n^2 .

Compressão de Dados

Descrição do problema

Padrões repetidos e fenômenos correlatos em palavras são conhecidos como o principal assunto em muitos aspectos das telecomunicações e na computação teórica, notavelmente em teoria dos códigos e em compressão de dados, na teoria das linguagens formais e no projeto e análise de algoritmos. Vários algoritmos eficientes têm sido desenvolvidos para detectar e explorar a presença de subpalavras repetidas e outros tipos de regularidades

em palavras. Em compressão de dados, tal subsequência repetida pode ser utilizada para reduzir o tamanho da sequência original, como por exemplo é feito nos esquemas freqüentes de compressão de dados.

Solução (por [Szp93a])

O problema de padrões repetidos é estudado probabilisticamente. Um parâmetro importante de uma árvore dos sufixos, que é essencial na análise e no projeto de algoritmos sobre palavras e de compressão de dados, é a *profundidade de um sufixo*. Uma análise probabilística dessa medida é então realizada.

Capítulo 6

Considerações finais

A árvore dos sufixos é, certamente, uma das estruturas de dados mais estudadas em processamento de palavras. A fundamentação teórica básica, obtida através do estudo de um conjunto de definições e propriedades relativas a essa estrutura, permitiu-nos conhecer as principais características da árvore dos sufixos, motivando-nos a estudar os algoritmos para a sua construção.

No estudo desses algoritmos, iniciamos procurando compreender a essência do funcionamento do algoritmo seqüencial de McCreight [McC76], especialmente devido a sua utilidade no projeto de algoritmos paralelos para a construção da árvore dos sufixos. Os algoritmos seqüenciais restantes foram apenas mencionados.

O principal objetivo de nossa dissertação foi o estudo de algoritmos paralelos para construção da árvore dos sufixos. Dessa maneira, procuramos descrever detalhadamente o algoritmo apresentado em [AIL⁺88], que consideramos ser o mais importante dos algoritmos paralelos pioneiros com esse objetivo. Sua complexidade de tempo é $O(\log n)$, o trabalho realizado é $O(n \log n)$ e o espaço utilizado é $O(n^2)$. Um esquema alternativo de implementação pode reduzir o espaço utilizado para $O(n^{1+\delta})$, para qualquer $0 < \delta \leq 1$, com o acréscimo de um fator $(1/\delta)$ na complexidade de tempo do algoritmo.

Apesar de existirem diversos algoritmos seqüenciais para construção da árvore dos sufixos de complexidade de tempo linear, o número total de operações realizadas no algoritmo paralelo em [AIL⁺88] é proporcional a $n \log n$, independentemente do tamanho do alfabeto relacionado. Esse algoritmo está baseado no procedimento clássico de atribuição de nomes de [KMR72]. Esse procedimento produz uma estrutura de dados extremamente útil para algoritmos paralelos que trabalham com palavras, como mostrado em [CR91]. No entanto, essa técnica impossibilita que o trabalho desempenhado pelo algoritmo seja linear devido, basicamente, ao número de subpalavras, proporcional a $n \log n$, às quais o algoritmo deve atribuir nomes. Esse número teoricamente dita a complexidade de trabalho do algoritmo. O fator logarítmico se deve ao aumento das redundâncias entre as subpalavras, já que

ocorrem sobreposições a medida que elas se tornam maiores. O principal problema é que não existe uma forma consistente de selecionar somente uma subpalavra em um conjunto de subpalavras sobrepostas.

Tentamos mostrar uma visão geral do funcionamento dos algoritmos paralelos posteriores, que conseguem solucionar satisfatoriamente este *problema da quebra de simetria* e, portanto, são algoritmos de trabalho ótimo.

Vimos também de modo sucinto alguns algoritmos paralelos mais recentes. O algoritmo de Sahinalp e Vishkin [SV94] visa a quebra da simetria que ocorre no processo de atribuição de nomes, e assim reduzem o número de operações realizadas para linear. O algoritmo proposto tem complexidade de tempo $O(\log^2 n)$ e complexidade de trabalho $O(n)$. O espaço total utilizado continua sendo $O(n^{1+\delta})$, para qualquer $0 < \delta \leq 1$, como no algoritmo anterior de [AIL⁺88]. Hariharan [Har94] obteve o primeiro algoritmo paralelo de trabalho e de espaço ótimos, que constrói a árvore dos sufixos com complexidade de tempo $O(\log^4 n)$. Farach, Hariharan e Muthukrishnan [FHM], em um trabalho ainda não publicado, propõem um algoritmo paralelo aleatorizado para construção da árvore dos sufixos, de complexidade de tempo $O(\log n)$, complexidade de trabalho $O(n)$ e complexidade de espaço $O(n)$.

No contexto das aplicações, aproveitamos as estruturas de dados geradas no algoritmo paralelo para construção da árvore dos sufixos em [AIL⁺88] e tentamos descrever cuidadosamente o problema da busca da maior subpalavra repetida, o problema da busca dos identificadores de subpalavras e o problema do casamento de padrões, juntamente com a descrição de algoritmos paralelos para solucioná-los. Apresentamos ainda, idéias básicas de algumas outras aplicações, tais como a busca do menor ancestral comum, o casamento aproximado de padrões, a geração do autômato dos sufixos minimal e do autômato dos fatores minimal, a generalização da árvore dos sufixos para matrizes quadradas e a compressão de dados.

Acreditamos ter conseguido reunir em nossa dissertação os principais trabalhos desta área. Em especial, conseguimos mostrar algumas propriedades básicas relativas à árvore dos sufixos, como por exemplo, que a quantidade de vértices da árvore dos sufixos é no máximo linear no tamanho da palavra de entrada. Completamos ainda algumas demonstrações de tais propriedades que não se encontram na literatura. Diferentemente da descrição em [AIL⁺88], apresentamos os procedimentos paralelos contidos neste trabalho com a utilização da Estrutura Trabalho-Tempo de Apresentação de Algoritmos Paralelos. Isso certamente garante maior facilidade de análise e de compreensão desses procedimentos, além de evitar a preocupação com a alocação e o escalonamento de processadores. Detalhamos certos estágios do algoritmo em [AIL⁺88] (por exemplo, o procedimento *Árvore Esqueleto Modificada*), que estão descritos superficialmente no artigo original. Dentro do contexto das aplicações, conseguimos reunir um grande conjunto de problemas e de algoritmos, sequenciais e paralelos, para solucioná-los. Estas soluções são constituídas de um

primeiro estágio de pré-processamento, onde ocorre a construção da árvore dos sufixos. Descrevemos com detalhes especialmente o problema do casamento de padrões juntamente com um algoritmo paralelo eficiente para sua solução. A motivação para a escolha do casamento de padrões foi a certeza de que este é um dos problemas mais estudados em ciência da computação.

Terminado este nosso trabalho, sabemos que muito ainda há para se fazer. O estudo detalhado de algoritmos seqüenciais é, indubitavelmente, bastante relevante, já que eles podem fornecer princípios e técnicas para a construção de novos algoritmos paralelos. Sabemos, sobretudo, que os algoritmos paralelos mais importantes, em [SV94, Har94], não foram estudados com profundidade. É certo também que uma implementação de algum desses algoritmos paralelos completaria nosso trabalho. No entanto, encontramos uma dificuldade inicial na transformação e na implementação de um algoritmo descrito no modelo de memória compartilhada em um algoritmo para o modelo de memória distribuída. Enfatizamos, portanto, a necessidade de desenvolvimento de novos algoritmos paralelos para a construção da árvore dos sufixos, projetados para máquinas de memória distribuída.

Apêndice A

Implementação do Algoritmo AILSV

A implementação de um algoritmo paralelo para construção da árvore dos sufixos fazia parte de nossa proposta de dissertação inicial, apresentada em junho de 1995. Por sua descrição mais didática e bem detalhada, escolhemos o **algoritmo AILSV** para esse propósito. No primeiro semestre de 1995, enquanto aguardávamos a chegada de uma máquina paralela ao instituto, decidimos simular seqüencialmente todo processamento paralelo realizado no **algoritmo AILSV**, com o objetivo principal de compreender seu funcionamento e de realizar testes básicos. Um grupo de iniciação científica, orientado pelo Prof. Siang Wun Song e co-orientado por mim, composto pelos alunos do Bacharelado em Ciência da Computação do IME Daílton Santana de Almeida, Ernesto Nakamura e Mário Yasunori Yamauti, deu início a essa implementação. Uma parte do código dessa implementação, descrito em PASCAL, é apresentada na Seção A.1 e, na Seção A.2, apresentamos dois exemplos de execução.

A máquina paralela que esperávamos chegou no início do segundo semestre de 1995. A *PowerXplorer* da PARSYTEC é constituída por 16 nós, cada um com um processador PowerPC 601 para computação e um *transputer* T805 para comunicação, além de 32 Mbytes de memória, totalizando portanto 512 Mbytes de memória. A velocidade de pico teórica desta máquina é de mais de um bilhão de operações aritméticas por segundo (1,2 GFLOPS) e o sistema operacional é o PARIX. O modelo de computação paralela é o modelo de rede de interconexão. Dessa forma, um desafio inicial foi encontrado: esta máquina trabalha no modelo de rede enquanto que os algoritmos que estudamos pertencem ao modelo de memória compartilhada. Assim, devido à ausência de semelhanças entre estes modelos, deveríamos projetar um outro algoritmo paralelo para construção da árvore dos sufixos que se adequasse ao modelo de rede. Depois de algumas discussões infrutíferas decidimos abandonar esse projeto. A busca por uma implementação paralela eficiente provavelmente implicaria o projeto de um novo algoritmo paralelo, apropriado para máquinas paralelas de memória distribuída. Isso, além de ficar fora do escopo de nossa dissertação, demandaria um tempo adicional muito além do tempo já gasto.

A.1 Algoritmo Seqüencial

(* Construção da árvore esqueleto *)

Procedure ArvoreEsqueleto (s: string; var Raiz: unitPai);

var j, i, q: number;

begin

for i := NP downto 1 do (* em paralelo - simula vitória *)
BB[1,OrdSigma(s[i])] := i; (* do processador de menor número *)

for i := NP downto 1 do begin

j := BB[1,OrdSigma (s[i])];

DS[i,0] := j;

if j = i then begin (* j = vencedor(i) *)

Vert[i,0].Pai.x := Raiz.x;

Vert[i,0].Pai.y := Raiz.y;

Vert[i,0].Rotulo := DS[i,0];

Vert[i,0].Nfilhos := 0;

end;

end;

(* se $i > n$, convencionamos que $DS[i,q] = m+1$ *)

for q := 0 to n-1 do begin (* iterações da árvore-esqueleto *)

for i := NP downto 1 do begin

j := i + pow (2,q);

if j > m (* 1 - 2 *)

then BB[DS[i,q] , m+1] := i

else BB[DS[i,q] , DS[j,q]] := i;

end;

for i := NP downto 1 do begin

j := i + pow (2,q);

if j > m

then DS[i,q+1] := BB[DS[i,q] , m+1]

else DS[i,q+1] := BB[DS[i][q] , DS[j,q]]; (* 3 *)

end;

```

    for i := NP downto 1 do
        if DS[i,q+1] = i then begin
            Vert[i,q+1].Pai.x := DS[i,q];
            Vert[i,q+1].Pai.y := q;
            Vert[i,q+1].Rotulo := DS[i,q+1];
            Vert[i,q+1].Nfilhos := 0;
            (* no' criado tem zero filhos *)
            Inc (Vert[DS[i,q],q].Nfilhos);
            (* incrementa numero de filhos do no' pai *)
        end;
    for i := NP downto 1 do
        if Vert[DS[i,q],q].Nfilhos = 1 then begin
            Vert[i,q+1].Pai.x := Vert[DS[i,q],q].Pai.x;
            Vert[i,q+1].Pai.y := Vert[DS[i,q],q].Pai.y;
        end;
    end;
end;
end;

```

(* Construção da árvore esqueleto com rótulos modificados *)

Procedure ArvoreEsqueletoMRotuladaMDx;

```

var p: array [1..NP] of unitPai;
    j, t, ql: number;

```

begin

```

    for j := 1 to NP do begin
        p[j].x := j;
        p[j].y := n;
    end;

```

```

    for t := n downto 0 do
        for i := NP downto 1 do

```

```

            if ((Vert[p[i].x,p[i].y].Pai.x <> Raiz.x) or
                (Vert[p[i].x,p[i].y].Pai.y <> Raiz.y)) then begin
                ql := Vert[p[i].x,p[i].y].Pai.y;

```

```

        if ql = t-1 then begin
            j := Vert[p[i].x,p[i].y].Rotulo;
            Vert[p[i].x,p[i].y].MRotulo.x := j + pow (2,ql);
            Vert[p[i].x,p[i].y].MRotulo.y := pow (2,p[i].y) -
                                                pow (2,ql);
            p[i].x := Vert[p[i].x,p[i].y].Pai.x;
            p[i].y := Vert[p[i].x,p[i].y].Pai.y;
        end;
    end
else if t = 0 then begin
    Vert[p[i].x,p[i].y].MRotulo.x :=
        Vert[p[i].x,p[i].y].Rotulo;
    Vert[p[i].x,p[i].y].MRotulo.y := pow (2,p[i].y);
    p[i].x := Raiz.x;
    p[i].y := Raiz.y;
end;

end;

(* ----- Programa principal ----- *)

begin
    Raiz.x := 0;
    Raiz.y := 0;

    s := ParamStr(1);
    for i := 1 + length (s) to m do s := s + '#';
    writeln ('Teste: ', s);

    ArvoreEsqueleto (s, Raiz);

    writeln;
    writeln ('Matriz DS');
    writeln;
    for i := 1 to NP do begin
        for j := 0 to n do
            write (DS[i,j]:4);
        writeln;
    end;
end;

```



```

writeln;
writeln ('Arvore Esqueleto');
writeln;
for i := 1 to NP do begin
    for j := 0 to n do
        write (Vert[i,j].Rotulo:5, '(' ,Vert[i,j].Pai.x:3, ', ',
                Vert[i,j].Pai.y:3, ') ');
    writeln;
end;

ArvoreEsqueletoMRotuladaMDx;

writeln;
writeln ('Arvore Esqueleto M-Rotulada');
writeln;
for i := 1 to NP do begin
    for j := 0 to n do
        write ((' ', Vert[i,j].MRotulo.x:3, ' ',
                Vert[i,j].MRotulo.y:3, ') ');
    writeln;
    for j := 0 to n do
        write ((' ',Vert[i,j].Pai.x:3, ', ',Vert[i,j].Pai.y:3, ') ');
    writeln;
    writeln;
end;

end.

```

A.2 Exemplos

Exemplo 1:

Teste: abbabbabbabbaab#

Matriz DS

1	1	1	1	1
2	2	2	2	2
2	3	3	3	3
1	1	1	1	4
2	2	2	2	5
2	3	3	3	6
1	1	1	7	7
2	2	2	8	8
2	3	3	9	9
1	1	1	10	10
2	2	11	11	11
2	3	12	12	12
1	13	13	13	13
1	1	14	14	14
2	15	15	15	15
16	16	16	16	16

Arvore Esqueleto

1(0, 0)	1(1, 0)	1(1, 1)	1(1, 2)	1(1, 3)
2(0, 0)	2(2, 0)	2(2, 1)	2(2, 2)	2(2, 3)
0(0, 0)	3(2, 0)	3(3, 1)	3(3, 2)	3(3, 3)
0(0, 0)	0(0, 0)	0(0, 0)	0(0, 0)	4(1, 3)
0(0, 0)	0(0, 0)	0(0, 0)	0(0, 0)	5(2, 3)
0(0, 0)	0(0, 0)	0(0, 0)	0(0, 0)	6(3, 3)
0(0, 0)	0(0, 0)	0(0, 0)	7(1, 2)	7(1, 2)
0(0, 0)	0(0, 0)	0(0, 0)	8(2, 2)	8(2, 2)
0(0, 0)	0(0, 0)	0(0, 0)	9(3, 2)	9(3, 2)
0(0, 0)	0(0, 0)	0(0, 0)	10(1, 2)	10(1, 2)
0(0, 0)	0(0, 0)	11(2, 1)	11(2, 1)	11(2, 1)
0(0, 0)	0(0, 0)	12(3, 1)	12(3, 1)	12(3, 1)
0(0, 0)	13(1, 0)	13(1, 0)	13(1, 0)	13(1, 0)
0(0, 0)	0(0, 0)	14(1, 1)	14(1, 1)	14(1, 1)
0(0, 0)	15(2, 0)	15(2, 0)	15(2, 0)	15(2, 0)
16(0, 0)	16(0, 0)	16(0, 0)	16(0, 0)	16(0, 0)

Arvore Esqueleto M-Rotulada

```

( 1, 1) ( 2, 1) ( 3, 2) ( 5, 4) ( 9, 8)
( 0, 0) ( 1, 0) ( 1, 1) ( 1, 2) ( 1, 3)

( 2, 1) ( 3, 1) ( 4, 2) ( 6, 4) ( 10, 8)
( 0, 0) ( 2, 0) ( 2, 1) ( 2, 2) ( 2, 3)

( 0, 0) ( 4, 1) ( 5, 2) ( 7, 4) ( 11, 8)
( 0, 0) ( 2, 0) ( 3, 1) ( 3, 2) ( 3, 3)

( 0, 0) ( 0, 0) ( 0, 0) ( 0, 0) ( 12, 8)
( 0, 0) ( 0, 0) ( 0, 0) ( 0, 0) ( 1, 3)

( 0, 0) ( 0, 0) ( 0, 0) ( 0, 0) ( 13, 8)
( 0, 0) ( 0, 0) ( 0, 0) ( 0, 0) ( 2, 3)

( 0, 0) ( 0, 0) ( 0, 0) ( 0, 0) ( 14, 8)
( 0, 0) ( 0, 0) ( 0, 0) ( 0, 0) ( 3, 3)

( 0, 0) ( 0, 0) ( 0, 0) ( 0, 0) ( 11, 12)
( 0, 0) ( 0, 0) ( 0, 0) ( 1, 2) ( 1, 2)

( 0, 0) ( 0, 0) ( 0, 0) ( 0, 0) ( 12, 12)
( 0, 0) ( 0, 0) ( 0, 0) ( 2, 2) ( 2, 2)

( 0, 0) ( 0, 0) ( 0, 0) ( 0, 0) ( 13, 12)
( 0, 0) ( 0, 0) ( 0, 0) ( 3, 2) ( 3, 2)

( 0, 0) ( 0, 0) ( 0, 0) ( 0, 0) ( 14, 12)
( 0, 0) ( 0, 0) ( 0, 0) ( 1, 2) ( 1, 2)

( 0, 0) ( 0, 0) ( 0, 0) ( 0, 0) ( 13, 14)
( 0, 0) ( 0, 0) ( 2, 1) ( 2, 1) ( 2, 1)

( 0, 0) ( 0, 0) ( 0, 0) ( 0, 0) ( 14, 14)
( 0, 0) ( 0, 0) ( 3, 1) ( 3, 1) ( 3, 1)

( 0, 0) ( 0, 0) ( 0, 0) ( 0, 0) ( 14, 15)
( 0, 0) ( 1, 0) ( 1, 0) ( 1, 0) ( 1, 0)

```


(0, 0) (0, 0) (0, 0) (0, 0) (16, 14)
 (0, 0) (0, 0) (1, 1) (1, 1) (1, 1)

(0, 0) (0, 0) (0, 0) (0, 0) (16, 15)
 (0, 0) (2, 0) (2, 0) (2, 0) (2, 0)

(0, 0) (0, 0) (0, 0) (0, 0) (16, 16)
 (0, 0) (0, 0) (0, 0) (0, 0) (0, 0)

Exemplo 2:

Teste: abbabababbbaba##

Matriz DS

1	1	1	1	1
2	2	2	2	2
2	3	3	3	3
1	1	4	4	4
2	3	3	5	5
1	1	4	6	6
2	3	7	7	7
1	1	8	8	8
2	2	9	9	9
2	2	2	10	10
2	3	3	11	11
1	1	12	12	12
2	3	13	13	13
1	14	14	14	14
15	15	15	15	15
15	16	16	16	16

Arvore Esqueleto

1(0, 0)	1(1, 0)	1(1, 1)	1(1, 1)	1(1, 1)
2(0, 0)	2(2, 0)	2(2, 1)	2(2, 2)	2(2, 2)
0(0, 0)	3(2, 0)	3(3, 1)	3(3, 2)	3(3, 2)
0(0, 0)	0(0, 0)	4(1, 1)	4(4, 2)	4(4, 2)
0(0, 0)	0(0, 0)	0(0, 0)	5(3, 2)	5(3, 2)
0(0, 0)	0(0, 0)	0(0, 0)	6(4, 2)	6(4, 2)
0(0, 0)	0(0, 0)	7(3, 1)	7(3, 1)	7(3, 1)
0(0, 0)	0(0, 0)	8(1, 1)	8(1, 1)	8(1, 1)
0(0, 0)	0(0, 0)	9(2, 1)	9(2, 1)	9(2, 1)
0(0, 0)	0(0, 0)	0(0, 0)	10(2, 2)	10(2, 2)
0(0, 0)	0(0, 0)	0(0, 0)	11(3, 2)	11(3, 2)
0(0, 0)	0(0, 0)	12(1, 1)	12(1, 1)	12(1, 1)
0(0, 0)	0(0, 0)	13(3, 1)	13(3, 1)	13(3, 1)
0(0, 0)	14(1, 0)	14(1, 0)	14(1, 0)	14(1, 0)
15(0, 0)	15(15, 0)	15(15, 0)	15(15, 0)	15(15, 0)
0(0, 0)	16(15, 0)	16(15, 0)	16(15, 0)	16(15, 0)

Arvore Esqueleto M-Rotulada

(1, 1)	(2, 1)	(0, 0)	(0, 0)	(3, 14)
(0, 0)	(1, 0)	(1, 1)	(1, 1)	(1, 1)
(2, 1)	(3, 1)	(4, 2)	(0, 0)	(6, 12)
(0, 0)	(2, 0)	(2, 1)	(2, 2)	(2, 2)
(0, 0)	(4, 1)	(5, 2)	(0, 0)	(7, 12)
(0, 0)	(2, 0)	(3, 1)	(3, 2)	(3, 2)
(0, 0)	(0, 0)	(6, 2)	(0, 0)	(8, 12)
(0, 0)	(0, 0)	(1, 1)	(4, 2)	(4, 2)
(0, 0)	(0, 0)	(0, 0)	(0, 0)	(9, 12)
(0, 0)	(0, 0)	(0, 0)	(3, 2)	(3, 2)
(0, 0)	(0, 0)	(0, 0)	(0, 0)	(10, 12)
(0, 0)	(0, 0)	(0, 0)	(4, 2)	(4, 2)
(0, 0)	(0, 0)	(0, 0)	(0, 0)	(9, 14)
(0, 0)	(0, 0)	(3, 1)	(3, 1)	(3, 1)

$(0, 0) (0, 0) (0, 0) (0, 0) (10, 14)$
 $(0, 0) (0, 0) (1, 1) (1, 1) (1, 1)$

$(0, 0) (0, 0) (0, 0) (0, 0) (11, 14)$
 $(0, 0) (0, 0) (2, 1) (2, 1) (2, 1)$

$(0, 0) (0, 0) (0, 0) (0, 0) (14, 12)$
 $(0, 0) (0, 0) (0, 0) (2, 2) (2, 2)$

$(0, 0) (0, 0) (0, 0) (0, 0) (15, 12)$
 $(0, 0) (0, 0) (0, 0) (3, 2) (3, 2)$

$(0, 0) (0, 0) (0, 0) (0, 0) (14, 14)$
 $(0, 0) (0, 0) (1, 1) (1, 1) (1, 1)$

$(0, 0) (0, 0) (0, 0) (0, 0) (15, 14)$
 $(0, 0) (0, 0) (3, 1) (3, 1) (3, 1)$

$(0, 0) (0, 0) (0, 0) (0, 0) (15, 15)$
 $(0, 0) (1, 0) (1, 0) (1, 0) (1, 0)$

$(15, 1) (0, 0) (0, 0) (0, 0) (16, 15)$
 $(0, 0) (15, 0) (15, 0) (15, 0) (15, 0)$

$(0, 0) (0, 0) (0, 0) (0, 0) (17, 15)$
 $(0, 0) (15, 0) (15, 0) (15, 0) (15, 0)$

Apêndice B

Lista de Símbolos

Símbolo	Significado	Primeira Ocorrência
Adj	vetor para busca de adjacências	38
BB	matriz para atribuição de nomes	38
D_x	árvore esqueleto	36
$D^{(0)}$	árvore dos sufixos invertida	50
$D^{(k)}$	k -ésimo refinamento de $m-D_x$	47
$\hat{D}^{(k)}$	refinamento $D^{(k-1)}$ no Passo 1	49
DS	matriz dos descritores	38
F	conjunto dos vértices folha	23
G_x	DAWG associado a x	83
$\mathcal{L}(y)$	locus da palavra y	73
$\mathcal{P}$	função pai	22
$\mathcal{P}^k$	k -composição de $\mathcal{P}$	22
R_u	vetor de ramificações de u	52
$\mathcal{S}$	função sucessor	20
$\tilde{\mathcal{S}}$	função sucessor estendida	20
$\mathcal{T}$	árvore t -ária	19
$\mathcal{T}^i$	sub-árvore de raiz v_i	20
T	árvore <i>trie</i>	23
$\mathcal{T}_x$	árvore dos sufixos esticada	24
T_x	árvore dos sufixos	18
V	conjunto de vértices	19
$ V $	tamanho da árvore	27
$Vért$	matriz dos vértices	38
$\mathcal{W}$	função passeio	22

Símbolo	Significado	Primeira Ocorrência
$d(v)$	grau do vértice v	20
$h(v)$	altura do vértice v	27
$h(v_0) = h(T)$	altura da árvore T de raiz v_0	27
i_y	quantas palavras y ocorrem em x	66
$m-D_x$	árvore esqueleto modificada	44
$ord_\Sigma(x_i)$	ordem do símbolo x_i em Σ	38
$su f_i$	i -ésimo sufixo	37
v_0	vértice raiz	19
x_i	i -ésimo símbolo da palavra x	05
$x(i)$	i -subpalavra	63
$x_{\{i\}}$	i -ésima palavra	66
$ w $	tamanho da palavra w	05
ϵ	palavra vazia	05
η	núcleo	63
λ	vértice vazio	20
σ_j	j -ésimo símbolo de Σ	20
τ	trabalho	10
ψ	função bijetora de F em $[1, n]$	23
Σ	alfabeto	05
Σ^*	todas as palavras sobre Σ	06
$\cdot$	concatenação de palavras	05
$\#$	símbolo especial	33

Bibliografia

- [AI87] A. Apostolico and C. Iliopoulos. Parallel log-time construction of suffix trees. CSD TR 632, Purdue University, Department of Computer Science, September 1987.
- [AIL⁺88] A. Apostolico, C. Iliopoulos, G.M. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3(3):347–365, 1988.
- [Aku93] T. Akutsu. A linear time pattern matching algorithm. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Proceedings, 4th Annual Symposium on Combinatorial Pattern Matching, CPM 93*, volume 684 of *Lecture Notes in Computer Science*, pages 1–10, Padova, Italy, June 1993. Springer-Verlag.
- [AN95] A. Andersson and S. Nilsson. Efficient implementation of suffix trees. *Software-Practice and Experience*, 25(2):129–141, 1995.
- [BEH89] A. Blumer, A. Ehrenfeucht, and D. Haussler. Average sizes of suffix trees and DAWGs. *Discrete Applied Mathematics*, 24:37–45, 1989.
- [CL90] W. Chang and E. Lawler. Approximate string matching in sublinear expected time. In *Proceedings, 31th IEEE Conference on the Foundations of Computer Science, FOCS 90*, pages 116–124, St. Louis, Missouri, USA, October 1990.
- [CL94] W. Chang and E. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, 1994.
- [Col88] R. Cole. Parallel merge sort. *SIAM J. Computing*, 17(4):770–785, 1988.
- [CR90] M. Crochemore and W. Rytter. Parallel construction of minimal suffix and factor automata. *Information Processing Letters*, 35(3):121–128, 1990.
- [CR91] M. Crochemore and W. Rytter. Usefulness of the Karp-Miller-Rosenberg algorithm in parallel computations on strings and arrays. *Theoretical Computer Science*, 88(1):59–82, 1991.

- [DSR92] L. Devroye, W. Szpankowski, and B. Rais. A note on the height of suffix trees. *SIAM J. Computing*, 21(1):48–53, 1992.
- [FHM] M. Farach, R. Hariharan, and S. Muthukrishnan. Optimal parallel suffix tree construction. (*trabalho em andamento*).
- [Fre60] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–500, 1960.
- [GG88] Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *Journal of Complexity*, 4(1):33–72, 1988.
- [Gia93] R. Giancarlo. The suffix of a square matrix, with applications. In *Proceedings, 4th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 93*, pages 402–411, Austin, Texas, USA, January 1993.
- [Gia95] R. Giancarlo. A generalization of the suffix tree to square matrices, with applications. *SIAM J. Computing*, 24(3):520–562, 1995.
- [Har94] R. Hariharan. Optimal parallel suffix tree construction. In *Proceedings, 26th Annual ACM Symposium on the Theory of Computing, STOC 94*, pages 290–299, Montreal, Quebec, Canada, May 1994. ACM Press.
- [HT84] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Computing*, 13(2):338–355, 1984.
- [JáJ92] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, Inc., Maryland, 1992.
- [JS91] P. Jacquet and W. Szpankowski. What can we learn about suffix trees from independent tries? In F. Dehne, J.-R. Sock, and N. Santoro, editors, *Proceedings, 2nd Workshop on Algorithms and Data Structures, WADS 91*, volume 519 of *Lecture Notes in Computer Science*, pages 228–239, Ottawa, Canada, August 1991. Springer-Verlag.
- [KD95] S. R. Kosaraju and A. L. Delcher. Large-scale assembly of DNA strings and space-efficient construction of suffix trees. In *Proceedings, 27th Annual ACM Symposium on the Theory of Computing, STOC 95*, pages 169–177, Las Vegas, Nevada, USA, June 1995. ACM Press. (preliminary version).
- [KMR72] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings, 4th Annual ACM Symposium on the Theory of Computing, STOC 72*, pages 125–136, Denver, Colorado, USA, May 1972. ACM Press.

- [Knu73] D. E. Knuth. *The Art of Computer Programming*, volume 3 : Sorting and Searching of *Computer Science and Information Processing*, chapter 6 : Searching, pages 481–505. Addison-Wesley Publishing Company, Inc., Oslo, Norway, 1973.
- [Kos94] S. R. Kosaraju. Real-time pattern matching and quasi-real-time construction of suffix trees. In *Proceedings, 26th Annual ACM Symposium on the Theory of Computing, STOC 94*, pages 310–316, Montreal, Quebec, Canada, May 1994. ACM Press. (preliminary version).
- [LO92] R. Lin and S. Olariu. A fast cost-optimal parallel algorithm for the lowest common ancestor problem. *Parallel Computing*, 18(5):511–516, 1992.
- [LP81] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall software series. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.
- [LSV87] G. M. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree. In T. Ottmann, editor, *Proceedings, 14th Annual International Colloquium on Automata, Languages and Programming, ICALP 87*, volume 267 of *Lecture Notes in Computer Science*, pages 314–325, Karlsruhe, Federal Republic of Germany, July 1987. Springer-Verlag.
- [LV89] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989.
- [Mah92] H. M. Mahmoud. *Evolution of random search trees*. Wiley - Interscience series in Discrete Mathematics and Optimization. John Wiley & Sons, Inc., Washington, DC, USA, 1992.
- [McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of ACM*, 23(2):262–272, 1976.
- [SV81] Y. Shiloach and U. Vishkin. Finding the maximum, merging and sorting in a parallel model of computation. *Journal of Algorithms*, 2(1):88–102, 1981.
- [SV88] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Computing*, 17(6):1253–1262, 1988.
- [SV94] S. C. Sahinalp and U. Vishkin. Symmetry breaking for suffix tree construction. In *Proceedings, 26th Annual ACM Symposium on the Theory of Computing, STOC 94*, pages 300–309, Montreal, Quebec, Canada, May 1994. ACM Press. (extended abstract).

- [Szp92] W. Szpankowski. Probabilistic analysis of generalized suffix trees. In A. Apostolico, Z. Galil, and U. Manber, editors, *Proceedings, 3rd Annual Symposium on Combinatorial Pattern Matching, CPM 92*, volume 644 of *Lecture Notes in Computer Science*, pages 1–14, Tucson, Arizona, USA, April 1992. Springer-Verlag. (extended abstract).
- [Szp93a] W. Szpankowski. Asymptotic properties of data compression. *IEEE Trans. on Information Theory*, 39(5):1647–1659, 1993.
- [Szp93b] W. Szpankowski. A generalized suffix tree and its (un)expected asymptotic behaviors. *SIAM J. Computing*, 22(6):1176–1198, 1993.
- [Ukk93] E. Ukkonen. Approximate string-matching over suffix trees. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Proceedings, 4th Annual Symposium on Combinatorial Pattern Matching, CPM 93*, volume 684 of *Lecture Notes in Computer Science*, pages 228–242, Padova, Italy, June 1993. Springer-Verlag.
- [Ukk95] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
- [Vis94] U. Vishkin. Can parallel algorithms enhance serial implementation? In H. J. Siegel, editor, *8th International Parallel Processing Symposium*, pages 376–385, Cancún, Mexico, April 1994. IEEE Computer Society Press.
- [Wei73] P. Weiner. Linear pattern matching algorithms. In *Proceedings, IEEE 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, Iowa, USA, October 1973.