O Autômato dos Sufixos

Ricardo Ueda Karpischek

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. Imre Simon**

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

-São Paulo, setembro de 1993-

O Autômato dos Sufixos

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Ricardo Ueda Karpischek e aprovada
pela comissão julgadora.

São Paulo, 3 de setembro de 1993.

Banca examinadora:

- Prof. Dr. Imre Simon (orientador) - IME-USP
- Prof. Dr. Yoshiharu Kohayakawa - IME-USP
- Prof. Dr. Nívio Ziviani - ICEX-UFMG

Agradecimentos

Aos meus pais, que muito zelaram para que eu estudasse,

Ao Prof. Imre Simon, de quem tanto aprendi nesses três anos em que pude contar com a sua valiosa orientação,

À Banca Examinadora, pela cuidadosa leitura e pelas inúmeras correções,

À Prfa. Nami, ao Lázaro, ao Éder, à Prfa. Clélia e ao Júlio, pela revisão do texto,

A tantos que nos ajudaram com referências, principalmente o Prof. Romeu Guimarães,

Ao Arnaldo e à Wânia, por colocarem em ordem essas formidáveis ferramentas de pesquisa que são as estações de trabalho,

Ao Flávio, por ter providenciado as cópias da versão provisória,

Ao Eduardo e ao Carlos, pela ajuda na implementação do índice do sistema de documentação,

Às Professoras Iracema, Carmem e Vera pelos primórdios da minha formação acadêmica,

E ao CNPq, pela bolsa,

o meu *sincero agradecimento.*

Ricardo Ueda Karpischek

Resumo

Elaboramos uma prova da correção e linearidade no tempo do algoritmo de construção do Autômato dos Sufixos devido a Blumer et alli. Ressaltamos o fato de que esse algoritmo também obtém a Árvore dos Sufixos do reverso da entrada, fato conhecido mas pouco explorado. Demos ainda dois resultados negativos que obtivemos para o problema da representação do Autômato dos Sufixos em espaço linear preservando a linearidade no tempo de construção (exige-se independência no tamanho do alfabeto).

Desenvolvemos uma implementação espaço-econômica do algoritmo de refinamento de Manber e Myers para a construção do Vetor dos Sufixos. Ela consome 2/3 da memória usada por aquela apresentada pelos autores. Fizemos alguns comentários sobre a informatização do Dicionário de Oxford, e apresentamos um algoritmo de Gonnet et alli, desenvolvido durante aquele projeto de processamento de textos. Concluímos a dissertação com um capítulo contendo alguns exemplos da atual interação entre Ciência da Computação e Biologia Molecular.

Todos os algoritmos apresentados foram implementados e testados exaustivamente. Alguns dos programas feitos serão distribuídos em breve.

Abstract

A proof of correctness and linear time complexity construction of the Suffix Automaton due to Blumer et alli is given. It is stressed that this algorithm also obtains the Suffix Tree for the reverse of the input word, a known but little explored fact. The representation problem of the Suffix Authomaton in linear space whithout loss of the linearity in construction time (independence on the alphabet size is required) is treated, and two negative results are presented.

We develop a space-economical implementation of the refinament algorithm due to Manber and Myers for Suffix Vector construction. It requires 2/3 of the memory used by that presented by the authors. A note is given on the informatization of the Oxford English Dictionary, and an algorithm developed by Gonnet et alli along that large-scale text-processing project is described. Finally, a chapter with examples of the interface between Computer Science and Molecular Biology closes this dissertation.

All algorithms presented here were implemented and exhaustively tested. Some of the programs developed will soon be distributed.

Conteúdo

Introdução

O Autômato dos Sufixos é especialmente interessante pelas notáveis propriedades que possui. Dentre elas, a mais importante provavelmente é que o seu tamanho (número de estados mais número de transições) é linear no tamanho da entrada, independentemente do tamanho do alfabeto. A isso deve-se adicionar o fato de que ele pode ser construído por um algoritmo tempo-linear, publicado por Blumer et alli há pouco menos de dez anos.

O esforço de pesquisa que culminou com o trabalho que referimos pode ser visto como uma continuação de outro, que tem como tema a Árvore dos Sufixos, para a qual conhece-se um algoritmo tempo-linear de construção desde 1974, devido a Weiner. Essas duas estruturas estão intimamente relacionadas por uma dualidade conhecida mas pouco explorada que desempenha um papel fundamental quando se deseja construir uma das duas.

No capítulo 1, que é o mais importante da nossa dissertação, apresentamos uma demonstração da correção do Algoritmo de construção do Autômato dos Sufixos de que falamos acima. Estudamos também o problema da representação do Autômato dos Sufixos em tempo linear (naturalmente isso só faz sentido quando se deseja manter a linearidade no tempo da construção independentemente do tamanho do alfabeto considerado), e apresentamos dois resultados negativos que obtivemos.

O Vetor dos Sufixos e a sua recente aplicação no projeto de informatização do Dicionário de Oxford são o tema do capítulo 2. Apresentamos três algoritmos para a sua construção. O primeiro é nosso, e roda em tempo linear, dentro de certas hipóteses. O segundo é um algoritmo de refinamento bastante elegante, devido a Manber e a Myers. Baseados numa implementação desses autores que consome memória equivalente a $3|x|$ variáveis inteiras, onde x é a palavra para a qual deseja-se construir o vetor dos sufixos, desenvolvemos uma outra, que consome o equivalente a $2|x|$ variáveis inteiras. Uma tal implementação já existia, mas nunca foi publicada. Descrevemos

também um interessante algoritmo de ordenação externa obtido por Gonnet et alli durante a informatização do Dicionário de Oxford.

Finalmente, no capítulo 3 descrevemos três problemas provenientes da Biologia Molecular (cálculo de homologias, detecção de repetições aproximadas e identificação de genes) e alguns métodos computacionais que não são propriamente soluções para eles, mas sim heurísticas que geram sugestões. Dentro do possível, procuramos conservar o vínculo desses métodos com as estruturas baseadas em sufixos de que falamos nos outros capítulos.

Dentre os programas que desenvolvemos ao longo do mestrado, selecionaremos alguns que, após alguns pequenos reparos serão colocados à disposição da comunidade acadêmica. Entre esses programas encontram-se ferramentas para a construção das estruturas de que tratamos na dissertação e outros, baseados nelas, que calculam diversas estatísticas sobre palavras. Temos também um pequeno sistema para a confecção de índices baseado naquele desenvolvido para o Dicionário de Oxford, e um programa para o cálculo do autômato reduzido de um vocabulário finito de grandes proporções.

Capítulo 1

O Autômato dos Sufixos

O autômato reduzido incompleto que reconhece a linguagem formada pelos sufixos de uma palavra tem tamanho linear no comprimento dessa palavra, e pode ser construído em tempo linear no mesmo parâmetro, independentemente do tamanho do alfabeto [1]. Tanto esse autômato quanto variações suas podem ser empregados no problema da busca de padrões, seja diretamente [7], seja de forma mais elaborada [8], incluindo busca com erros [15]. Também podem ser aplicados no cálculo eficiente de uma série de estatísticas sobre uma palavra, algumas delas úteis em biologia molecular[6]. Existe ainda uma métrica sobre seqüências computável em tempo linear através do autômato dos sufixos [10].

Definiremos o autômato dos sufixos e provaremos a correção do algoritmo dado em [4, 7] que o constrói. Descreveremos uma de suas relações com a Árvore dos Sufixos e mostraremos como construí-la a partir do autômato dos sufixos [2]. Nossa apresentação é completa no sentido de não remeter a resultados que se encontram em outros textos, mas não fomos além do que já podia, de uma forma ou de outra, ser encontrado em [4], a menos de uma ou duas observações acidentais. É nosso o que está na última seção, referente ao consumo de memória, problema que tem favorecido o uso de estruturas menos sofisticadas como o Vetor dos Sufixos, já empregado com sucesso em

[1] Para descongestionar o texto, "linear no comprimento de x" subentenderá de ora em diante que a linearidade independe do tamanho do alfabeto, ainda que, por ênfase, às vezes digamo-lo explicitamente

[2] Historicamente, a Árvore dos Sufixos antecedeu o Autômato dos Sufixos, mas como o nosso interesse principal é o Autômato, invertemos essa ordem.

dois projetos de grande porte, e que descreveremos no capítulo 2.

1.1 Conjuntos de Términos

Usando a unicidade do autômato reduzido definiremos o autômato dos sufixos, e em seguida associaremos os seus estados com *conjuntos de términos*, que representam as *ocorrências*, em uma palavra, de cada um dos seus fatores. Essa associação será o primeiro passo no levantamento de uma série de propriedades que utilizaremos na prova da correção do algoritmo de construção do autômato dos sufixos.

1.1.1 Sufixos e Fatores

Sejam A um alfabeto finito [3] e $x = x_1x_2 \ldots x_{|x|}$ uma palavra sobre A ($x_i \in A$, $i = 1, 2, \ldots, |x|$). Os *sufixos* de x são as palavras da forma $x_ix_{i+1} \ldots x_{|x|}$, onde $1 \leq i \leq |x|$, mais a palavra vazia λ, e os *sufixos próprios* de x são os sufixos de x, exceto x (portanto λ é um sufixo próprio de x se $x \neq \lambda$). Por sua vez, os *fatores* de x são as palavras da forma $x_ix_{i+1} \ldots x_j$, onde $1 \leq i, j \leq |x|$ (nossa convenção é que se $i > j$ então $x_ix_{i+1} \ldots x_j$ denota λ). Os conjuntos formados pelos sufixos e pelos fatores de x serão denotados respectivamente por $\mathrm{Suf}(x)$ e $\mathrm{Fat}(x)$. Mais adiante faremos menção também dos *prefixos* de x, isto é, das palavras da forma $x_1x_2 \ldots x_j$ onde $1 \leq j \leq |x|$, mais a palavra vazia λ. O conjunto dos prefixos de x será denotado por $\mathrm{Pref}(x)$.

Trabalharemos com autômatos finitos determinísticos onde os rótulos das transições – exceto quando tratarmos da árvore dos sufixos – são letras de A. Nossos autômatos serão *não necessariamente completos* (por brevidade, *incompletos* ou simplesmente *AFDIs*), isto é, dados um estado α do AFDI $\mathcal{C}$ e uma letra b, pode não existir em $\mathcal{C}$ uma transição com origem α e rótulo b. O conjunto dos estados (finais) de $\mathcal{C}$ será denotado por Q_c (F_c). Consideraremos também que as transições de $\mathcal{C}$ *pertencem* às suas respectivas origens, e por isso diremos *as transições de α em $\mathcal{C}$*, quando quisermos nos referir às transições de $\mathcal{C}$ que têm o estado α como origem.

Dada uma letra $b \in A$ e um estado α do autômato $\mathcal{C}$, se α tiver (em $\mathcal{C}$) a transição com b, então denotaremos por $(\alpha b)_c$ o destino dessa transição.

[3] Faremos referência ao longo do texto (subentendendo o seu significado) a essa e a muitas outras notações que introduziremos aos poucos.

A mesma convenção valerá para $u \in A^*$ no lugar de b e *passeio* no lugar de transição.

Finalmente, para cada estado $\alpha \in Q_c$, $P_c(\alpha)$ denotará o conjunto das palavras soletradas pelos passeios de $\mathcal{C}$ com origem no estado inicial e término em α, enquanto $S_c(\alpha)$ denotará o conjunto das palavras soletradas pelos passeios com origem em α e término em algum estado final de $\mathcal{C}$.

Os AFDI's que reconhecem $\mathrm{Suf}(x)$ com número mínimo de estados formam uma classe de isomorfismo. Sem prejuízos à precisão, qualquer um deles será chamado de *o Autômato dos Sufixos de* x (também chamado DAWG – Directed Acyclic Word Graph). Assim, ao dizermos que $\mathcal{A}$ é o autômato dos sufixos de x (expressão que abreviaremos para $\mathcal{A} = \mathrm{AutSuf}(x)$), estaremos querendo dizer que $\mathcal{A}$ é um AFDI que reconhece $\mathrm{Suf}(x)$ com quantidade mínima de estados.

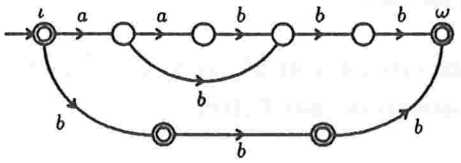


Figura 1.1: O Autômato dos Sufixos de *aabbb*

A minimalidade do autômato dos sufixos $\mathcal{A}$ de x garante uma série de propriedades elementares. Em primeiro lugar ele é *acessível e co-acessível* (ou seja, para cada estado α de $\mathcal{A}$, os conjuntos $P_{\mathcal{A}}(\alpha)$ e $S_{\mathcal{A}}(\alpha)$ são não-vazios); juntando a isso o fato da linguagem reconhecida ser finita vem que ele é acíclico. Se os estados α e β são distintos então $S_{\mathcal{A}}(\alpha) \neq S_{\mathcal{A}}(\beta)$, e portanto há um único estado sem transições, que será chamado de *o sorvedouro*. Dele ser co-acessível e determinístico vem que a família $\{P_{\mathcal{A}}(\alpha); \alpha \in Q_{\mathcal{A}}\}$ é uma partição de $\mathrm{Fat}(x)$. Em particular, o autômato que obtemos fazendo todos os seus estados finais [4] reconhece $\mathrm{Fat}(x)$.

Alguns dos estados de $\mathcal{A}$ receberão nomes especiais. Por enquanto introduziremos dois deles: ι e ω que serão, respectivamente, o estado inicial e o

[4]Esse novo autômato em geral têm estados redundantes. Reduzindo-o obtemos o *Autômato dos Fatores* de x, de que não trataremos mas que é o tema central dos trabalhos que introduziram o Autômato dos Sufixos [4, 7].

sorvedouro de $\mathcal{A}$. É fácil ver que $\omega = (\iota x)_{\mathcal{A}}$, pois, se $(\iota x)_{\mathcal{A}}$ tivesse alguma transição, então na linguagem reconhecida por $\mathcal{A}$ haveria uma palavra de comprimento maior que $|x|$.

1.1.2 Classificação via Términos

Os estados do autômato dos sufixos de x classificam os fatores de x de acordo com as suas *ocorrências* em x, no sentido em que precisaremos na proposição 2. Antes disso necessitamos de uma definição: diremos que $j \in \{0, 1, \ldots, |x|\}$ é *término de* $u \in \mathrm{Fat}(x)$ *em* x se tivermos $u = x_i x_{i+1} \ldots x_j$ para algum $i \in \{1, 2, \ldots, j\}$, ou se $u = \lambda$. O conjunto dos términos de u em x será denotado por $t_x(u)$. Assim, por exemplo, $t_x(\lambda) = \{0, 1, \ldots, |x|\}$ e $t_x(x) = \{|x|\}$. Seja $\alpha = (\iota u)_{\mathcal{A}}$. Então para cada $j \in \{0, 1, \ldots, |x|\}$ temos que $j \in t_x(u)$ se e somente se o sufixo $x_{j+1} x_{j+2} \ldots x_{|x|}$ pertencer a $S_{\mathcal{A}}(\alpha)$. Daqui concluímos de imediato a seguinte proposição:

Proposição 1 *Para cada estado α de $\mathcal{A}$, se $y, z \in P_{\mathcal{A}}(\alpha)$ então $t_x(y) = t_x(z)$. Esse valor comum será denotado por $t_{\mathcal{A}}(\alpha)$.*

Note que, dado um fator u de x ou um estado α de $\mathcal{A}$, então $t_x(u)$ e $t_{\mathcal{A}}(\alpha)$ são conjuntos não vazios. Esse fato estará implícito em muitos dos raciocínios que faremos ao longo deste capítulo.

Proposição 2 *A aplicação $t_{\mathcal{A}} : \alpha \in Q_{\mathcal{A}} \mapsto t_{\mathcal{A}}(\alpha)$ é injetora.*

Prova: Suponha que os estados $\alpha, \beta \in Q_{\mathcal{A}}$ satisfaçam $t_{\mathcal{A}}(\alpha) = t_{\mathcal{A}}(\beta)$. Então $S_{\mathcal{A}}(\alpha) = \{x_j x_{j+1} \ldots x_{|x|}; j \in t_{\mathcal{A}}(\alpha)\} = \{x_j x_{j+1} \ldots x_{|x|}; j \in t_{\mathcal{A}}(\beta)\} = S_{\mathcal{A}}(\beta)$, donde vem $\alpha = \beta$, visto que $\mathcal{A}$ é reduzido.

Os conjuntos de términos exibem ainda outras propriedades elementares que serão exploradas mais adiante e que daremos a seguir:

Proposição 3 *Sejam u e v fatores de x. Se $u \in \mathrm{Suf}(v)$ então $t_x(v) \subseteq t_x(u)$. Se $t_x(u)$ e $t_x(v)$ tiverem interseção não vazia, então $t_x(u) \subseteq t_x(v)$ ou $t_x(v) \subseteq t_x(u)$. Se a inclusão $t_x(v) \subseteq t_x(u)$ for estrita, então u é sufixo de v.*

Prova: A primeira afirmação é evidente, pois todo término de v em x também o é de u. Quanto à segunda, de u e v terem um término comum em x segue

que u é sufixo de v ou v é sufixo de u, e basta aplicar agora a primeira parte. Finalmente, de $t_x(v) \subseteq t_x(u)$ tiramos que u e v têm um término comum em x, e, como antes, que u é sufixo de v ou v é sufixo de u. Ora, essa segunda possibilidade é impossível em virtude da inclusão ser estrita.

1.2 A Árvore Esticada

Identificaremos um subautômato do autômato dos sufixos que irá orientar o nosso estudo do algoritmo de construção do autômato dos sufixos, através das noções de fator esticado e de ancestral. Em seguida definiremos a árvore dos sufixos e estudaremos o vínculo existente entre ela e o autômato dos sufixos.

1.2.1 Caminhos mais longos

Dado um estado α de $\mathcal{A}$, está bem definido o caminho mais longo de ι a α, pois caso contrário na linguagem reconhecida por $\mathcal{A}$ haveria duas palavras diferentes de mesmo comprimento, uma vez que o autômato dos sufixos é determinístico e co-acessível. Com isso estamos autorizados a fixar x_α como sendo *a palavra mais longa em* $P_{\mathcal{A}}(\alpha)$.

As palavras do conjunto $\{x_\alpha; \alpha \in Q_{\mathcal{A}}\}$ serão chamadas de *os fatores esticados de* x. Todo fator de x que não for esticado em x será dito *frouxo* em x. É imediato (proposições 1 e 2) que se $u \in \mathrm{Fat}(x)$ for esticado em x então dentre todos os fatores de x cujo conjunto de términos em x é $t_x(u)$, u é o de maior comprimento.

Considere os caminhos de $\mathcal{A}$ que partem do estado inicial e soletram os fatores esticados de x. Definimos as *transições esticadas de* $\mathcal{A}$ como sendo as transições de $\mathcal{A}$ que pertencem a algum desses caminhos (as restantes serão chamadas *frouxas*.

Todo estado α de $\mathcal{A}$ (exceto o inicial) é destino de pelo menos uma transição esticada. De fato, a última transição do maior caminho de ι a α é esticada. Suponha que α seja destino de *duas* transições esticadas de $\mathcal{A}$. Então ao menos uma delas (que iremos chamar de t) não pertence ao caminho mais longo C_1 de ι a α. Sejam C_2 um caminho em $\mathcal{A}$ que parte de ι e usa t, e β o seu destino. Então C_2 não é o caminho mais longo de ι a β em $\mathcal{A}$, visto que a partir dele podemos construir um maior, trocando o trecho

que vai de ι a α por C_1. Ora, isso contradiz a definição de transição esticada, e portanto a hipótese inicial era falsa.

Chamando de *árvore* os autômatos determinísticos acíclicos onde cada estado, exceto o inicial (ou *raiz*), é destino de exatamente uma transição, temos a proposição que segue:

Proposição 4 *O autômato resultante de $\mathcal{A}$ pela remoção das transições frouxas é uma árvore com raiz ι, que será chamada de a árvore esticada de $\mathcal{A}$.*

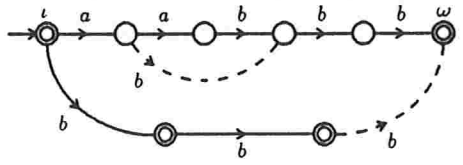


Figura 1.2: A Árvore esticada do Autômato dos Sufixos de *aabbb*. As transições frouxas estão tracejadas.

Provemos ainda que a palavra x_α não é apenas a palavra mais longa de $P_{\mathcal{A}}(\alpha)$, mas guarda com esse conjunto uma relação mais forte:

Proposição 5 *$P_{\mathcal{A}}(\alpha)$ é um intervalo superior de* $\mathrm{Suf}(x_\alpha)$*, isto é, existe um inteiro $k \geq 0$ tal que $P_{\mathcal{A}}(\alpha) = \{u \in \mathrm{Suf}(x_\alpha); |u| \geq k\}$. Se $\alpha \neq \iota$, então $k-1$ é o comprimento do maior sufixo próprio de x_α que é fator esticado de x.*

Prova: Dado $u \in P_{\mathcal{A}}(\alpha)$, então $t_x(u) = t_x(x_\alpha)$ (proposição 1), e como $|u| \leq |x_\alpha|$ deduzimos pela proposição 3 que u é sufixo de x_α. Seja v um sufixo de x_α com $|v| \geq |u|$. Naturalmente u é sufixo de v, e portanto valem as inclusões $t_x(x_\alpha) \subseteq t_x(v) \subseteq t_x(u)$, e como já tínhamos $t_x(u) = t_x(x_\alpha)$ segue que $t_x(v) = t_x(x_\alpha)$, donde pela proposição 2 vem $v \in P_{\mathcal{A}}(\alpha)$.

Se $\alpha \neq \iota$, então $\lambda \notin P_{\mathcal{A}}(\alpha)$, e portanto $k > 0$ e o sufixo u de tamanho $k-1$ de x_α está bem definido. Se provarmos que ele é fator esticado de x estamos feitos. Seja $\beta = (\iota u)_{\mathcal{A}}$. Ora, de $\beta \neq \alpha$ deduz-se que $t_x(u)$ contém propriamente $t_x(x_\alpha)$, e portanto pela proposição 3 vem que x_β é sufixo de x_α. Aplicando agora a primeira parte temos que é impossível valer $|x_\beta| > |u|$, donde segue $u = x_\beta$.

1.2.2 Caracterização dos Fatores Esticados

Dado um fator esticado u de x, se u for sufixo próprio de $v \in \mathrm{Fat}(x)$, então $t_x(v) \subseteq t_x(u)$ mas não vale a igualdade, pois se valesse então u não seria a palavra mais longa com conjunto de términos $t_x(u)$, contra a hipótese de u ser esticada. Por outro lado, se $u \in \mathrm{Fat}(x)$ for frouxo em x, então u é sufixo próprio de x_α, onde $\alpha = (\iota u)_A$ (proposição 5). Portanto $t_x(u) = t_x(x_\alpha)$, e provamos a primeira caracterização:

Proposição 6 *$u \in \mathrm{Fat}(x)$ é fator esticado de x se e somente se para todo $v \in \mathrm{Fat}(x)$ do qual u é sufixo próprio valer $|t_x(v)| < |t_x(u)|$.*

Corolário 7 *$v \in \mathrm{Fat}(x)$ é fator esticado de x se e somente se v for prefixo de x ou se existirem letras $b, c \in A$ distintas tais que bv e cv são fatores de x. Em particular, λ, x e prefixos de fatores esticados são fatores esticados de x.*

Prova: Se v for prefixo de x, então para todo $u \in \mathrm{Fat}(x)$ do qual v é sufixo próprio, temos que $|v| \in t_x(v) \setminus t_x(u)$, e portanto $|t_x(u)| < |t_x(v)|$. Se existirem letras $b, c \in A$ distintas tais que av e bv são fatores de x, então tomando $u \in \mathrm{Fat}(x)$ tal que v é sufixo próprio de u, podemos supor sem perda de generalidade que bv não é sufixo de u, e temos $|t_x(u)| \leq |t_x(v) \setminus t_x(bv)| < |t_x(v)|$.

Suponha agora que v seja fator esticado de x mas não prefixo de x. Então existe uma letra $b \in A$ tal que $bv \in \mathrm{Fat}(x)$. Pela proposição 6 vem que $|t_x(bv)| < |t_x(v)|$, e portanto existe um término j de v que não é término de bv. Como $j > |v|$, temos que a letra $c = x_{j-|v|}$ está bem definida, e pela escolha de j segue que $c \neq b$.

1.2.3 Sufixos encadeados

Seja u um fator esticado de x. Se $u \neq \lambda$, então u tem ao menos um sufixo próprio que é fator esticado de x, visto que λ é fator esticado de x (corolário 7). O maior sufixo próprio de u que for fator esticado de x será chamado de *o tronco* de u em x, enquanto u será um dos seus *ramos*. Suponha que α e β sejam os estados de $\mathcal{A}$ que satisfazem $x_\alpha = u$ e $x_\beta = v$. Chamaremos β de *o pai* em $\mathcal{A}$ de α (o tronco de λ em x e o pai em $\mathcal{A}$ de ι devem ser considerados indefinidos).

Seja $m > 0$, o maior inteiro para o qual é possível definir a seqüência $\alpha_1, \alpha_2, \ldots, \alpha_m$ de estados de $\mathcal{A}$, onde $\alpha_1 = \alpha$ e α_i é o pai em $\mathcal{A}$ de α_{i-1} para $1 < i \leq m$ (naturalmente $\alpha_m = \iota$). Para cada $i \in \{0, 1, \ldots, m\}$, o estado α_i será chamado de *o i-ésimo ancestral de α em $\mathcal{A}$.*

Proposição 8 *Dados $\alpha, \beta \in Q_{\mathcal{A}}$, β é ancestral de α em $\mathcal{A}$ se e somente se x_β for sufixo de x_α.*

Prova: A ida segue da transitividade da relação "é sufixo de". A volta é uma indução elementar: suponha $|x_\alpha| > |x_\beta| \geq 0$. Seja γ o pai de α em $\mathcal{A}$. Se $\gamma = \beta$ não há o que fazer. Senão x_β é sufixo de x_γ. Como $|x_\gamma| < |x_\alpha|$, por hipótese β é ancestral de γ em $\mathcal{A}$ e daí segue o que queremos.

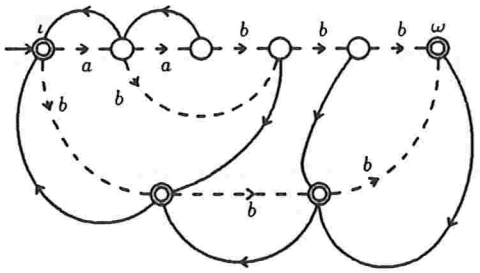


Figura 1.3: Os pais no Autômato dos Sufixos de *aabbb*

Corolário 9 *Dados $\alpha, \beta \in Q_{\mathcal{A}}$, β é ancestral de α em $\mathcal{A}$ se e somente se $t_{\mathcal{A}}(\alpha) \subseteq t_{\mathcal{A}}(\beta)$.*

Corolário 10 *Os ancestrais de ω em $\mathcal{A}$ são os estados finais de $\mathcal{A}$.*

Proposição 11 *Dados $\alpha \in Q_{\mathcal{A}}$, um ancestral β de α em $\mathcal{A}$, e $b \in A$, suponha que α tenha em $\mathcal{A}$ a transição com b. Então β tem em $\mathcal{A}$ a transição com b e $(\beta b)_{\mathcal{A}}$ é ancestral de $(\alpha b)_{\mathcal{A}}$ em $\mathcal{A}$.*

Prova: Da hipótese vem que existe um término j de x_α em x tal que $j < |x|$ e $x_{j+1} = b$. Ora, j é término de x_β em x (pois x_β é sufixo de x_α), e portanto $x_\beta b \in \mathrm{Fat}(x)$, donde segue que β tem em $\mathcal{A}$ a transição com b. A justificativa da segunda afirmação é semelhante: $x_\beta a$ é sufixo de $x_\alpha b$ e aplica-se o corolário 9.

1.2.4 A Árvore dos Sufixos

Um AFDI $\mathcal{S}$ com rótulos em A^+ será dito *f-determinístico* se os rótulos de cada par de transições com origem comum diferirem nas suas primeiras letras, isto é, se forem da forma bu e cv com $b, c \in A$ distintas. No caso de $\mathcal{S}$ ser uma árvore, diremos que cada estado α de $\mathcal{S}$ *realiza* a palavra soletrada pelo único caminho em $\mathcal{S}$ da raiz até α. Como os rótulos são palavras, podem existir prefixos de palavras em $|\mathcal{S}|$ que não são realizados por nenhum estado de $\mathcal{S}$.

Proposição 12 *Sejam $\mathcal{S}$ uma árvore f-determinística e u uma palavra sobre A. Se existirem letras $b, c \in A$ distintas tais que ub e uc são prefixos de palavras em $|\mathcal{S}|$, então u é realizada por algum estado de $\mathcal{S}$.*

Prova: Sejam $v, w \in |\mathcal{S}|$ tais que ub é prefixo de v e uc é prefixo de w. Dentre os estados de $\mathcal{S}$ que estão simultaneamente nos caminhos C_v e C_w que soletram v e w a partir da raiz, tome β maximizando o comprimento da palavra realizada. Seja z a palavra realizada por β em $\mathcal{S}$. Naturalmente z é prefixo de u. Pela escolha de β, esse estado tem que ser origem de duas transições distintas, uma em C_v e outra em C_w, e se z fosse prefixo próprio de u, então os rótulos dessas duas transições não difeririam nas suas primeiras letras, contra a hipótese.

Diremos que a árvore f-determinística $\mathcal{S}$ tem número de estados mínimo se não existir uma árvore f-determinística com o mesmo comportamento de $\mathcal{S}$ e com menos estados que $\mathcal{S}$. Daremos uma condição suficiente para a minimalidade e provaremos a unicidade da árvore f-determinística com número de estados mínimo. Isso irá permitir-nos definir a *Árvore dos sufixos de x* como sendo a árvore f-determinística que reconhece $\mathrm{Suf}(x)$ com número mínimo de estados [5].

Corolário 13 *É suficiente para que a árvore f-determinística $\mathcal{S}$ tenha número de estados mínimo que cada estado de $\mathcal{S}$ que não é final seja origem de pelo menos duas transições.*

[5] Supondo que a última letra de x ocorra uma única vez em x, (isto é, que $t_x(x_{|x|}) = \{|x|\}$) e que $|x| > 1$, a árvore dos sufixos de x pode ser caracterizada como a árvore f-determinística cujo comportamento é $\mathrm{Suf}(x)$ e onde cada estado ou é folha ou tem pelo menos duas saídas (omite-se a prova). Essa caracterização muitas vezes é tomada como *definição* da árvore dos sufixos. Se, entretanto, $x_{|x|}$ ocorrer mais de uma vez em x, então pode não existir uma árvore nas condições dessa caracterização.

Prova: Tome uma árvore f-determinística $\mathcal{S}'$ que satisfaça $|\mathcal{S}'| = |\mathcal{S}|$. Seja α um estado de $\mathcal{S}$ e u a palavra por ele realizada. Se α for final então claramente existe um estado β de $\mathcal{S}'$ que realiza u. Se α não for final, então da hipótese vem que existem letras b, c distintas tais que ub e uc são prefixos de palavras em $|\mathcal{S}|$, e pela proposição 12 existe um estado β de $\mathcal{S}'$ que realiza u.

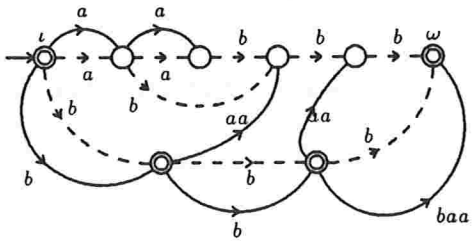


Figura 1.4: A Árvore dos Sufixos de *bbbaa* com as transições do Autômato dos Sufixos de *aabbb* tracejadas.

A recíproca do corolário 13 não é verdadeira apenas num caso muito especial. Suponha que o estado α de $\mathcal{S}$ seja não final e tenha grau de saída 1. Sejam β e u respectivamente o destino e rótulo dessa transição. Suponha também que α não seja a raiz. Então α é destino em $\mathcal{S}$ de exatamente uma transição. Sejam γ a origem dessa transição e v o seu rótulo. Se trocarmos o destino da transição de γ para β e o seu rótulo para vu, removendo de $\mathcal{S}$ o estado α juntamente com a sua transição (ou seja, se *contrairmos* a transição de α), a linguagem reconhecida não será alterada, mas o número de estados diminui[6], resultado que enunciaremos na forma de um corolário:

Corolário 14 *Se a árvore f-determinística $\mathcal{S}$ tiver número de estados mínimo, então, exceto pela raiz, todo estado de $\mathcal{S}$ que não for final é origem de pelo menos duas transições.*

Corolário 15 *Duas árvores f-determinísticas com mesmo comportamento e número mínimo de estados são isomorfas.*

[6]A iteração desse processo conduz, naturalmente, à árvore dos sufixos.

Prova: Sejam $\mathcal{S}'$ e $\mathcal{S}$ conforme o enunciado. Mostraremos, como na demonstração do corolário 13, que existe uma função que a cada estado α de $\mathcal{S}'$ associa um estado β de $\mathcal{S}$ preservando a palavra realizada. Se α for final em $\mathcal{S}'$, a escolha de β é clara. Suponha então que α não seja final. Se α for origem de pelo menos duas transições de $\mathcal{S}'$ então repete-se o raciocínio da demonstração citada. Se α não for origem de pelo menos duas transições de $\mathcal{S}'$, então α é necessariamente a raiz de $\mathcal{S}'$, e portanto escolhemos para β a raiz de $\mathcal{S}$.

Agora provaremos a forte conexão existente entre o autômato dos sufixos de x e a árvore dos sufixos do reverso x^R de x. Para isso, considere o autômato com rótulos em A^+ cujos estados são os mesmos que os de $\mathcal{A}$, e onde existe uma transição de α para β se e somente se β for o pai de α em $\mathcal{A}$. No caso da transição existir, definimos como sendo o seu rótulo o reverso u^R da palavra u que satisfaz $ux_\beta = x_\alpha$. Tomaremos ι para raiz e definimos que α é final se, e somente se, x_α for prefixo de x. Esse autômato será denotado por $\mathcal{T}^R$.

Proposição 16 *O autômato $\mathcal{T}^R$ é a árvore dos sufixos de x^R.*

Prova: É imediato que em $\mathcal{T}^R$ o grau de entrada do estado inicial é zero e o dos outros estados é um. Suponha que α seja origem de duas transições de $\mathcal{T}^R$, e sejam u e v os seus rótulos. Então, $u^R x_\alpha$ e $v^R x_\alpha$ são fatores esticados de x. Seja z o maior sufixo comum a $u^R x_\alpha$ e a $v^R x_\alpha$, que é fator esticado de x. Naturalmente, x_α é sufixo de z, mas não próprio, pois caso contrário nenhuma das duas transições de α em $\mathcal{T}^R$ de que estamos falando existiria. Daí vem que a primeira letra de u é diferente da primeira letra de v, e portanto $\mathcal{T}^R$ é uma árvore f-determinística.

Mostraremos agora por indução no comprimento do caminho em $\mathcal{T}^R$ que vai da raiz até α, que a palavra realizada por α é o reverso de x_α. Supondo $\alpha \neq \iota$, sejam β a origem da única transição de $\mathcal{T}^R$ que atinge α, e u o seu rótulo. Por hipótese, a palavra realizada por β em $\mathcal{T}^R$ é o reverso de x_β, e de $u^R x_\beta = x_\alpha$ vem o que queremos.

Agora podemos provar que $|\mathcal{T}^R| = \mathrm{Suf}(x^R)$. De fato, se $u \in |\mathcal{T}^R|$, temos que u^R é prefixo de x, e portanto sufixo de x^R. Reciprocamente, se u for sufixo de x^R então u é prefixo de x e, portanto, fator esticado de x (corolário 7), e se α for o estado de $\mathcal{A}$ que satisfaz $x_\alpha = u$, então α é final em $\mathcal{T}^R$ e realiza u em $\mathcal{T}^R$.

Para concluir, seja α um estado de $\mathcal{T}^R$ que não é final em $\mathcal{T}^R$. Então, x_α não é prefixo de x, e pelo corolário 7 existem letras $b, c \in A$ distintas com $bx_\alpha, cx_\alpha \in \mathrm{Fat}(x)$. Portanto, α é origem de pelo menos duas transições de $\mathcal{T}^R$, uma com rótulo da forma bu e outra com rótulo da forma cv, com $b \neq c$. Aplicando agora o corolário 13 temos que $\mathcal{T}^R$ é a árvore dos sufixos de x^R.

A relação entre o autômato dos sufixos de x e a árvore dos sufixos de x^R intensifica-se pelo fato de que o algoritmo que constrói $\mathcal{A}$ constrói ao mesmo tempo $\mathcal{T}^R$. De fato, os pais em $\mathcal{A}$ serão obtidos como subproduto da construção de $\mathcal{A}$, e a partir de $\mathcal{A}$ e dos pais em $\mathcal{A}$ obtém-se automaticamente $\mathcal{T}^R$ (falaremos disso mais cuidadosamente em 1.4.4) [7].

1.3 A Construção incremental

Seja a uma letra de A, $y = xa$ e $\mathcal{B} = \mathrm{AutSuf}(y)$. A partir da noção de árvore esticada estudaremos a *diferença* entre os autômatos $\mathcal{B}$ e $\mathcal{A} = \mathrm{AutSuf}(x)$, a fim de, mais adiante, provarmos a correção do algoritmo de construção do autômato dos sufixos (ele obtém efetivamente o autômato dos sufixos de cada prefixo de x). A primeira conseqüência do estudo dessa diferença serão limites superiores justos para o número de estados e de transições de $\mathcal{A}$.

1.3.1 Número de estados e de transições

Para todo fator u de x, está claro que $t_y(u) = t_x(u)$ ou $t_y(u) = t_x(u) \cup \{|y|\}$, o segundo caso ocorrendo se e somente se u for sufixo de y. Suponha que u seja sufixo próprio de v e que $|t_x(v)| < |t_x(u)|$. Se $|y|$ for término de v em y, então $|y|$ é também término de u em y, donde $|t_y(v)| < |t_y(u)|$ e com isso provamos a

Proposição 17 *Toda palavra esticada em x é esticada em y.*

Daí decorre que $\mathcal{B}$ não tem menos estados que $\mathcal{A}$. Como y é esticada em y, a diferença $|Q_\mathcal{B}| - |Q_\mathcal{A}|$ é pelo menos um. Provaremos agora que essa

[7] O único algoritmo que conhecemos para obter os pais em $\mathcal{A}$ em tempo linear em $|x|$ é o algoritmo de construção de $\mathcal{A}$. Assim, a única maneira que conhecemos para transformar $\mathcal{A}$ em $\mathcal{T}^R$ em tempo linear em $|x|$ é "jogue fora $\mathcal{A}$ e construa $\mathcal{T}^R$". Um comentário análogo vale para a transformação de $\mathcal{T}^R$ em $\mathcal{A}$.

diferença é no máximo dois [8]. Denotaremos por $s = s(x, a)$ o maior sufixo de y que é fator de x. Como λ é sufixo de y e fator de x, s está bem definida.

Proposição 18 *A palavra s é sempre esticada em y*

Prova: Seja u um fator de y tal que s é sufixo próprio de u. Se u for sufixo de y, então $|t_y(u)| = 1 < |t_x(s)| + 1 = |t_y(s)|$, e se u não for sufixo de y então $|t_y(u)| = |t_x(u)| \leq |t_x(s)| < |t_y(s)|$.

Proposição 19 *Se $u \in \mathrm{Fat}(x)$ é esticado em y mas não em x, então $y = s$.*

Prova: Seja $v \in \mathrm{Fat}(x)$ tal que u é sufixo próprio de v e $|t_x(v)| = |t_x(u)|$. Seja b a letra que satisfaz $bu \in \mathrm{Suf}(v)$. Pela proposição 5 vem que $t_x(bu) = t_x(u)$, e segue a cadeia $t_x(u) \subseteq t_x(bu) \subseteq t_y(bu) \subseteq t_y(u) \subseteq t_x(u) \cup \{|y|\}$. Como u é esticada em y, vem que $|t_y(bu)| < |t_y(u)|$, e portanto a penúltima inclusão da cadeia é própria e a última é uma igualdade, donde u é sufixo de y e bu não. Seja $d \in A$ a letra definida por $du \in \mathrm{Suf}(y)$. Então $d \neq b$, e como u é frouxa em x, segue $du \notin \mathrm{Fat}(x)$. Portanto todo sufixo de y com comprimento maior que $|u|$ não é fator de x.

Corolário 20 *$\mathcal{B}$ tem no mínimo um e no máximo dois estados a mais do que $\mathcal{A}$, e se $|x| \geq 2$ então $|x| + 1 \leq |Q_{\mathcal{A}}| \leq 2|x| - 1$.*

Prova: A primeira afirmação já foi provada, pois o número de estados do autômato dos sufixos de uma palavra é igual ao número de fatores esticados dessa mesma palavra. A segunda decorre do fato do autômato dos sufixos de qualquer palavra de comprimento dois ter exatamente três estados [9].

Por inspeção pode-se verificar que uma palavra de três letras tem cinco fatores esticados se e somente se for da forma bcc com $b \neq c$. Por outro lado, é imediato que se $x = bc^n$ com $n > 0$, então s é frouxa em x se e somente se $a = c$. Assim, o limite superior dado na proposição 20 é atingido se e somente se x for da forma bc^n. Quanto ao limite inferior $|x| + 1$, ele segue de $|Q_{\mathcal{A}}| \geq |\mathrm{Pref}(x)| = |x| + 1$, e é atingido se $x = b^n$.

[8] É interessante notar que a diferença entre os números de estados de $\mathcal{C} = \mathrm{AutSuf}(ax)$ e de $\mathcal{A}$ pode ser positiva ou negativa, e nos dois casos pode ter módulo proporcional a $|x|$.

[9] Existe uma estimativa [5] para o número médio de estados dos Autômatos dos Sufixos de palavras de comprimento n sobre um alfabeto fixo. Ela diz que esse número médio não se estabiliza, mas exibe um comportamento periódico de período crescente (e exponencial) em n.

Corolário 21 *Se* $|x| \geq 3$ *então* $\mathcal{A}$ *tem no máximo* $3|x| - 4$ *e no mínimo* $|x|$ *transições.*

Prova: Aqui seguimos [4]. Suponha inicialmente $|x| \geq 2$. Já sabemos que $\mathcal{A}$ tem no máximo $2|x| - 1$ estados, e portanto há no máximo $2|x| - 2$ transições esticadas. Contemos pois as frouxas. Suponha que as transições de α com b e de β com c em $\mathcal{A}$ sejam frouxas. Vamos *completar* $x_\alpha b$ e $x_\beta c$ para sufixos de x, isto é, escolher $u, v \in \mathrm{Suf}(x)$ tais que $z = x_\alpha bu$ e $w = x_\beta cv$ sejam sufixos de x (note que z é necessariamente distinto de λ e de x).

Se $\alpha \neq \beta$, então $z \neq w$. De fato, suponha $z = w$. Assumindo sem perda de generalidade que $|x_\alpha| < |x_\beta|$, vem que $x_\alpha b$ é prefixo de x_β, mas isso é impossível, pois $x_\alpha b$ é frouxa em x, ao passo que prefixos de fatores esticados de x são esticados em x (corolário 7). Se $\alpha = \beta$, então é claro que $z = w$ acarreta $b = c$.

Com isso provamos que o total de transições frouxas não pode superar $|x| - 1$, visto que x tem $|x| + 1$ sufixos dos quais deve-se desconsiderar dois (λ e $|x|$, conforme já observamos). Assim o total de transições de $\mathcal{A}$ não supera $3|x| - 3$.

Passemos agora ao caso $|x| \geq 3$. Se x não for da forma bc^n com $b \neq c$, então $\mathcal{A}$ tem no máximo $2|x| - 2$ estados, e portanto a repetição do raciocínio anterior leva-nos ao limite superior de $3|x| - 4$ transições. Se x for da forma bc^n, então $\mathcal{A}$ tem no máximo uma transição frouxa, pois x tem apenas um fator frouxo (b^n), e novamente obtemos $3|x| - 4$ como limite superior. Quanto ao limite inferior $|x|$, ele segue de $|Q_{\mathcal{A}}| \geq |x| + 1$.

Os limites obtidos no corolário anterior são justos. De fato, $bc^n d$ com b, c, d letras distintas e $n > 0$ atinge o superior, enquanto $x = b^n$ atinge o inferior.

1.3.2 O Intervalo Livre

Apoiados na proposição 17, consideraremos que os estados de $\mathcal{A}$ são também estados de $\mathcal{B}$ num sentido forte, a saber, que *para cada estado* α *de* $\mathcal{A}$, α *é estado de* $\mathcal{B}$ *e* $y_\alpha = x_\alpha$. Essa hipótese norteará o nosso estudo das *diferenças* entre as transições de $\mathcal{A}$ e $\mathcal{B}$, e por isso é importante não perder de vista que $\mathcal{A}$ e $\mathcal{B}$ são *representantes fixos* das classes de isomorfismo a que pertencem.

Definimos $\theta = (\iota s)_{\mathcal{A}}$ e $\omega' = (\iota y)_{\mathcal{B}}$ (ω' é o sorvedouro de $\mathcal{B}$ – note que ι é o estado inicial de $\mathcal{B}$). Se s for frouxa em x, então colocaremos $\theta' = (\iota s)_{\mathcal{B}}$

(se s for esticada em x, então deve-se considerar que o estado θ' *não está definido*). Os estados ω' e θ' (quando estiver definido) são, é claro, os *estados de $\mathcal{B}$ que não são estados de $\mathcal{A}$.*

Suponha que a ocorra em x. Então existe pelo menos um estado final de $\mathcal{A}$ que tem a transição com a (a saber, ι). De todos os estados finais de $\mathcal{A}$ que têm a transição com a, considere aquele cuja palavra esticada associada tem comprimento máximo. Esse estado será denotado por μ (se a não ocorrer em x, então deve-se considerar que o estado μ *não está definido*).

Proposição 22 *μ está definido em $\mathcal{A}$ se e somente se $s \neq \lambda$, e, estando μ definido, $s = x_\mu a$ e $(\mu a)_{\mathcal{A}} = \theta$.*

Prova: A primeira é imediata, pois $s = \lambda$ se e somente se a não ocorrer em x. Suponha μ definido. Em $\mathcal{A}$, por definição, μ é final e tem a transição com a. Portanto $x_\mu a$ é sufixo de y e também é fator de x. Suponha que $u = va$ seja sufixo de y e fator de x. Pela definição de μ, x_μ não pode ter comprimento menor do que x_α, onde $\alpha = (\iota v)_{\mathcal{A}}$, donde segue que $x_\mu a$ é o maior sufixo de y que é fator de x, que é o que queríamos provar.

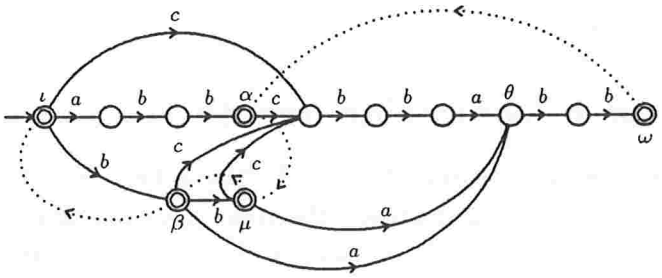


Figura 1.5: Os intervalos livre e frouxo para $x = abbcbbabb$. Os pais dos estados finais estão indicados pelas linhas pontilhadas. O intervalo livre é $\{\omega, \alpha\}$, e o frouxo é $\{\mu, \beta\}$.

Há dois subconjuntos de $F_{\mathcal{A}}$ que terão um papel importante (veja as proposições 25 e 26). Definiremos o *intervalo livre* $\mathcal{I} = \mathcal{I}(\mathcal{A}, a)$ de $\mathcal{A}$ como sendo o conjunto dos estados finais de $\mathcal{A}$ que não têm a transição com a. O intervalo livre nunca é vazio, pois o sorvedouro de $\mathcal{A}$ não tem transições. O nome *intervalo* deve-se à seguinte propriedade elementar, conseqüência trivial do corolário 10 e da proposição 11:

Proposição 23 *Existe um inteiro $m \geq 0$ tal que $\beta \in Q_{\mathcal{A}}$ pertence ao intervalo livre de $\mathcal{A}$ se e somente se β for o i-ancestral de ω em $\mathcal{A}$ para algum $i \in \{0, 1, \ldots, m\}$. Se μ estiver definido, então μ é exatamente o $(m+1)$-ésimo ancestral de ω em $\mathcal{A}$.*

O *intervalo frouxo* $\mathcal{F} = \mathcal{F}(\mathcal{A}, a)$ de $\mathcal{A}$ é, por sua vez, o conjunto dos estados finais α de $\mathcal{A}$ que satisfazem $(\alpha a)_{\mathcal{A}} = \theta$. Da proposição 22 vem que o intervalo frouxo só é vazio se $s = \lambda$. Ele só terá real importância para nós quando s for frouxa em x. Assim como no caso do intervalo livre, o seu nome deve-se à seguinte caracterização:

Proposição 24 *Suponha $s \neq \lambda$. Então existe $m \geq 0$ tal que o intervalo frouxo de $\mathcal{A}$ é formado pelos i-ésimos ancestrais de μ, onde $i \in \{0, 1, \ldots, m\}$.*

Prova: Já provamos (proposição 22) que $(\mu a)_{\mathcal{A}} = \theta$, e (proposição 23) que os estados finais de $\mathcal{A}$ que não são ancestrais de μ não têm a transição com a. É suficiente portanto mostrar que se α é ancestral de μ e $\alpha \notin \mathcal{F}$, então os ancestrais de α também não pertencem ao intervalo frouxo, mas isso segue de duas aplicações da proposição 11. De fato $(\alpha a)_{\mathcal{A}}$ é ancestral de θ e, por hipótese, distinto de θ. E se γ for ancestral de α então $(\gamma a)_{\mathcal{A}}$, por ser ancestral de $(\alpha a)_{\mathcal{A}}$, é distinto de θ.

1.3.3 Inclusão e Redirecionamento

Sejam α um estado de $\mathcal{A}$ e b uma letra de A. Suponha que α tenha em $\mathcal{B}$ a transição com b, e seja β o seu destino. Há três casos a considerar: α pode não ter em $\mathcal{A}$ a transição com b, ou, no caso de tê-la, o destino dessa transição pode ou não ser β. No primeiro caso diremos que a transição de α com b foi *incluída* em $\mathcal{B}$, no segundo que ela foi *preservada* em $\mathcal{B}$, e, no terceiro, que ela foi *redirecionada* em $\mathcal{B}$. Note que se α não tiver em $\mathcal{B}$ a transição com b, então α também não terá em $\mathcal{A}$ a transição com b (pois $\mathrm{Fat}(x) \subseteq \mathrm{Fat}(y)$), e portanto os três casos que demos esgotam todas as possibilidades.

Proposição 25 *A transição de $\alpha \in Q_{\mathcal{A}}$ com b é incluída em $\mathcal{B}$ se e somente se α pertencer ao intervalo livre de $\mathcal{A}$ e $b = a$.*

Prova: Suponha a inclusão. Então $x_a b$ é fator de y mas não de x, o que só pode ocorrer com sufixos de y. Portanto α é final em $\mathcal{A}$ e $b = a$. Reciprocamente, se $b = a$ e $\alpha \in F_{\mathcal{A}}$ não tiver a transição com b, então ela será incluída em $\mathcal{B}$, pois $x_a b$ é fator de y.

Proposição 26 *A transição de α com b é redirecionada em $\mathcal{B}$ se e somente se α pertencer ao intervalo frouxo de $\mathcal{A}$, s for frouxa em x e $b = a$.*

Prova: Suponha o redirecionamento. Seja $\beta = (\alpha b)_{\mathcal{A}}$. Então $x_\alpha b$ é sufixo de x_β, e podemos escrever a cadeia $t_x(x_\beta) = t_x(x_\alpha b) \subseteq t_y(x_\beta) \subseteq t_y(x_\alpha b) \subseteq t_x(x_\alpha b) \cup \{|y|\}$. A hipótese do redirecionamento leva-nos a concluir que a penúltima inclusão é própria, e portanto a antepenúltima é uma igualdade. Daí vem que $x_\alpha b$ é sufixo de y e que x_β não é. Em particular, α é final em $\mathcal{A}$ e $b = a$

Provemos agora que $\beta = \theta$. Por $x_\alpha b$ ser sufixo de y e fator de x, podemos concluir que é também sufixo de s. Daí vem que s e x_β têm términos comuns em x (qualquer um em $t_x(s)$), e como x_β não é sufixo de y, segue que s é sufixo próprio de x_β. Finalmente, da cadeia $t_x(x_\alpha b) \subseteq t_x(s) \subseteq t_x(x_\beta)$ deduzimos a igualdade entre os três conjuntos de términos, e segue que $\beta = \theta$ e também que s é frouxa em x.

A volta é imediata, pois de α pertencer ao intervalo frouxo de $\mathcal{A}$ e da igualdade $b = a$ vem que $t_y(x_\alpha b) = t_x(x_\alpha b) \cup \{|y|\} = t_x(s) \cup \{|y|\} = t_y(s)$. Assim o destino da transição de α com b em $\mathcal{B}$ é θ', que não é estado de $\mathcal{A}$, donde o redirecionamento.

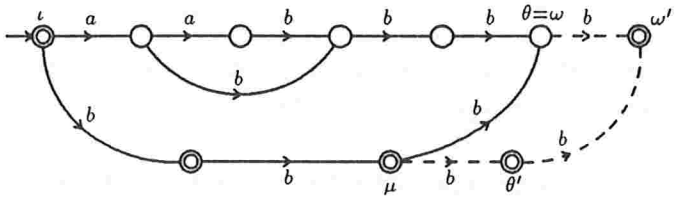


Figura 1.6: A construção do Autômato dos Sufixos de *aabbbb* a partir do Autômato dos Sufixos de *aabbb*. As transições incluídas ou redirecionadas estão tracejadas.

Corolário 27 *Todas as transições que sofrem redirecionamento têm em $\mathcal{A}$ destino θ, e, em $\mathcal{B}$, destino θ'.*

Podemos ainda inferir um limite superior para o número σ_y de transições de $\mathcal{A}$ que são redirecionadas em $\mathcal{B}$ em função do número de estados finais de $\mathcal{A}$, que será importante na prova da linearidade no tempo do algoritmo de construção do autômato dos sufixos:

Corolário 28 $\sigma_y \leq |F_{\mathcal{A}}| - |F_{\mathcal{B}}| + 2$.

Prova: Seja f a função que a cada estado $\alpha \in F_{\mathcal{A}}$ associa $(\alpha a)_{\mathcal{B}}$ (a proposição 25 implica que α tem em $\mathcal{B}$ a transição com a). Então f é sobrejetora a menos de ι, pois $\beta \in F_{\mathcal{B}} \setminus \{\iota\}$ é imagem por f de $(\iota u)_{\mathcal{A}} \in F_{\mathcal{A}}$ onde $ua = y_\beta$, e como todas as origens de transições redirecionadas são levadas por f num mesmo estado de $\mathcal{B}$ (corolário 27), vem que $|F_{\mathcal{B}}| - 1 \leq |F_{\mathcal{A}}| - \sigma_y + 1$ donde segue o enunciado.

1.3.4 As transições de $\mathcal{B}$

Resumiremos os resultados sobre inclusão e redirecionamento já obtidos numa descrição do conjunto das transições de $\mathcal{B}$. Antes disso entretanto é necessário estudarmos rapidamente as transições dos estados de $\mathcal{B}$ que não são estados de $\mathcal{A}$, isto é, ω' e, se s for frouxa em x, θ'. Sabemos que o primeiro não tem transições em $\mathcal{B}$. Quanto ao segundo, temos a

Proposição 29 *Se θ' estiver definido em $\mathcal{B}$, então as transições de θ' em $\mathcal{B}$ são as mesmas que as de θ em $\mathcal{B}$.*

Prova: Basta observar que $t_{\mathcal{B}}(\theta') \triangle t_{\mathcal{B}}(\theta) = \{|y|\}$, pois dessa igualdade segue que, para cada letra $b \in A$, $t_y(sb) = t_y(y_\theta b)$.

A importância da proposição 29 deve-se a que os resultados de inclusão e redirecionamento de que já dispomos permitem descrever as transições de θ em $\mathcal{B}$, e portanto agora podemos descrever o conjunto das transições de $\mathcal{B}$.

Definimos o *conjunto sem redirecionamentos* T_s como sendo a união entre o conjunto das transições de $\mathcal{A}$ e o conjunto das transições incluídas em $\mathcal{B}$. Se s for frouxa em x, então definimos o *conjunto com redirecionamentos* $T_c = (T_s \setminus \{\alpha \xrightarrow{a} \theta; \alpha \in \mathcal{F}\}) \cup \{\alpha \xrightarrow{a} \theta'; \alpha \in \mathcal{F}\} \cup \{\theta' \xrightarrow{b} \alpha; \theta \xrightarrow{b} \alpha \in T_s\}$.

Por definição, T_c difere de T_s por corrigir o destino das transições que são redirecionadas em $\mathcal{B}$ (proposição 26) e incluir as transições que θ' tem em $\mathcal{B}$ (proposição 29). Assim temos pelas próprias definições desses conjuntos a seguinte proposição:

Proposição 30 *O conjunto das transições de $\mathcal{B}$ é T_s se s for esticada em x ou T_c caso contrário.*

1.3.5 Os Pais em $\mathcal{B}$

Seja $u \neq \lambda$ um fator esticado de x, e v o seu tronco em x. Seja w o tronco de u em y. Se w for palavra esticada de x, então necessariamente $w = v$. De fato, se tivéssemos $w \neq v$ então, como ambos são sufixos de u, vem $|w| \neq |v|$, o que contradiria ou a definição de v ou a definição de w.

Assim, se $w \neq v$ então devemos concluir que w não é esticada em x, e de $|w| < |u| < |y|$ vem pela proposição 19 que $w = s$ e que s é frouxa em x. Podemos concluir também que $|v| < |w|$, e portanto pela proposição 5 temos que $(\iota u)_{\mathcal{A}} = (\iota s)_{\mathcal{A}} = \theta$. Resumindo, temos as duas proposições que seguem:

Proposição 31 *Dado $\alpha \in Q_{\mathcal{A}} \setminus \{\iota\}$, se $\alpha \neq \theta$ ou se s for esticada em x então o pai de α em $\mathcal{B}$ é o pai de α em $\mathcal{A}$.*

Proposição 32 *Se s for frouxa em x então o pai de θ em $\mathcal{B}$ é θ'.*

Passemos agora aos estados de $\mathcal{B}$ que não são estados de $\mathcal{A}$. É fácil ver que tanto ω' quanto θ' (desde que θ' esteja definido, é claro) são diferentes de ι, e portanto os seus pais em $\mathcal{B}$ estão definidos.

Proposição 33 *O pai de ω' em $\mathcal{B}$ é θ se s for esticada em x, ou θ' caso contrário*

Prova: Seja u sufixo próprio de y. Se $|u| > |s|$ então u não é fator de x, e portanto u ocorre uma única vez em y. Naturalmente u não é prefixo de y, e pelo corolário 7 vem que u é frouxa em y. Assim, o tronco de y em y é s e estamos feitos.

Proposição 34 *Suponha que s seja frouxa em x. Então o pai de θ' em $\mathcal{B}$ é o pai de θ em $\mathcal{A}$.*

Prova: Seja u o tronco de s em y. Da proposição 32 vem que u é sufixo (próprio) de x_θ. Assim, se v for o tronco de x_θ em x, então $|v| \geq |u|$. Da definição de u segue que a desigualdade só pode ser estrita se valer $|v| \geq |s|$, absurdo pois nesse caso a proposição 5 implica $(\iota v)_{\mathcal{A}} = \theta$. Portanto $v = u$ e é isso que queríamos provar.

1.4 O Algoritmo de Construção

Depois de dar um critério aritmético simples para decidir se uma transição de $\mathcal{A}$ é ou não esticada, apresentaremos o algoritmo de construção do Autômato dos Sufixos e provaremos a sua correção e linearidade no tempo, descrevendo explicitamente as estruturas de dados necessárias para isso. Mostraremos em seguida o que deve ser adicionado a esse algoritmo para que ele obtenha (ainda em tempo linear) a Árvore dos Sufixos do reverso de x com as transições de cada estado ordenadas lexicograficamente em relação aos seus rótulos (desde que se troque a *ordem alfabética* em A pela *ordem de aparecimento em* x).

1.4.1 Comprimentos dos caminhos mais longos

Na construção de $\mathcal{B}$ a partir de $\mathcal{A}$ deveremos decidir em algum momento se s é ou não esticada em x. Como λ é sempre esticada em x (corolário 7), basta estudar o caso $s \neq \lambda$. Lembrando que $s \neq \lambda$ equivale a μ estar definido (proposição 22), temos a seguinte proposição:

Proposição 35 *Se $s \neq \lambda$ então s é esticada em x se e só se $|x_\mu| + 1 = |x_\theta|$.*

Prova: Pela definição de θ, s é esticada em x se e somente se $s = x_\theta$, isto é, se e somente se o caminho mais longo de ι a θ em $\mathcal{A}$ tiver comprimento $|s|$. Por outro lado, a proposição 22 diz-nos que $s = x_\mu a$, e daí vem que o maior caminho de ι a μ em $\mathcal{A}$ tem comprimento $|s| - 1$, donde segue o enunciado.

Do ponto de vista da estrutura de dados utilizada para representar o autômato dos sufixos de x, a proposição 35 sugere-nos que cada estado α deverá possuir um campo para armazenar $|x_\alpha|$. Essa não é a maneira mais econômica de se implementar um teste que decida se s é ou não esticada em x. De fato, bastaria associar a cada transição um bit que informasse se ela é ou não esticada. Entretanto a informação $|x_\alpha|$ é bastante útil nas aplicações do autômato dos sufixos, e por isso é ela que usaremos.

1.4.2 O passo da construção

O algoritmo que segue constrói $\mathcal{B}$ a partir de $\mathcal{A}$. Por enquanto não entraremos em detalhes sobre a estrutura de dados utilizada para a representação desses autômatos (essa questão será abordada mais adiante): apenas indicaremos

quais operações ela deve permitir-nos realizar. Na descrição do algoritmo, a função **Cópia** aloca um estado e cria, para cada transição do argumento, uma transição do estado alocado com mesmos rótulo e destino. A expressão μa denota o destino da transição de μ com a.

```
1   função Passo(ι, ω, a)
2     NovoEstado(ω'); prof[ω'] ← prof[ω] + 1
3     α ← ω
4     Enquanto α não tiver a transição com a faça
5       crie a transição α -a-> ω'
6       se α ≠ ι então α ← pai[α]
7       senão pai[ω'] ← ι
8     se pai[ω'] estiver indefinido então
9       μ ← α; θ ← μa
10      se prof[μ] + 1 < prof[θ] então
11        θ' ← Cópia(θ); prof[θ'] ← prof[μ] + 1
12        enquanto αa = θ faça
13          redirecione a transição de α com a para θ'
14          se α ≠ ι então α ← pai[α]
15        pai[ω'] ← θ'; pai[θ'] ← pai[θ]; pai[θ] ← θ'
16      senão pai[ω'] ← θ
17    devolva(ω');
```

Algoritmo 1.1: A função **Passo**: ι, ω e a devem ser entendidos como parâmetros formais, ao passo que $\alpha, \omega', \mu, \theta$ e θ' como variáveis locais. A identação define os comandos compostos. O procedimento **NovoEstado** aloca um estado sem transições.

Devemos ser capazes de *alocar estados* inicialmente sem transições. Dado um estado α já alocado e uma letra $b \in A$, a estrutura de dados deve permitir-nos decidir se α tem ou não a transição com b, e, em caso afirmativo, obter o seu destino (que, subentende-se, é um estado já alocado). No caso em que α não tem a transição com b, poderemos adicioná-la, definindo o seu destino como sendo qualquer estado já alocado. No caso em que α tem a transição com b, devemos ser capazes de *redirecioná-la*, isto é, de redefinir

o seu destino. Finalmente, cada estado alocado deve possuir dois *atributos*, a saber, o *pai*, que deve poder ser qualquer estado alocado, e a *profundidade*, que é um inteiro; eles serão representados respectivamente por $pai[\alpha]$ e $prof[\alpha]$. Assim como as variáveis que utilizarmos, esses atributos devem ser considerados inicialmente indefinidos, e, uma vez definidos, poderão ser *redefinidos* se necessário [10].

Num determinado instante, sejam α e β dois estados alocados. Considere então o seguinte autômato: os seus estados são os estados alocados, e as suas transições são as transições representadas pela estrutura de dados; o seu estado inicial é α, e os seus estados finais são definidos recursivamente [11] por: β é final e, se γ é final e o seu atributo *pai* está definido, $pai[\gamma]$ é final. Esse autômato será referido por *o* (α, β)*-autômato representado.*

Suponha que o (α, β)-autômato representado seja $\mathcal{A}$ (isso significa, entre outras coisas, que α é ι e que β é ω). Diremos que os atributos *pai* e *profundidade* são *compatíveis com essa representação de* $\mathcal{A}$ se valer que, para cada estado γ, $prof[\gamma] = |x_\gamma|$ e que o atributo *pai* de γ está definido se e somente se $\gamma \neq \iota$ e, no caso de estar definido, $pai[\gamma]$ é o pai de γ em $\mathcal{A}$.

Proposição 36 *Suponha que o* (α, β)*-autômato representado é* $\mathcal{A}$ *e que os atributos* pai *e* profundidade *são compatíveis com essa representação de* $\mathcal{A}$*. Nessas condições, suponha que seja feita a chamada* **Passo**(α,β,a)*. Então a execução de* **Passo** *termina e, se* γ *for o seu valor de retorno, então, ao término da referida execução, o* (α, γ)*-autômato representado é* $\mathcal{B}$ *e os atributos* pai *e* profundidade *são compatíveis com essa representação de* $\mathcal{B}$*.*

A prova da proposição 36 será feita em etapas. Nas proposições que seguem, as hipóteses da proposição 36 estarão implicitamente feitas.

Naturalmente os valores dos *parâmetros formais* ι e ω são, respectivamente, os *estados* ι e ω de $\mathcal{A}$, e o valor do *parâmetro formal* a é precisamente a letra a, e por isso será desnecessário especificar se estamos falando do *parâmetro* ι ou do *estado* ι, a mesma observação valendo para a e ω.

[10] Faremos ainda duas últimas hipóteses que não são essenciais. Uma é que os estados contam com uma *marca* (um estado recém-alocado está, por hipótese, desmarcado), e outra é que pode-se percorrer todos os estados alocados, isto é, existe um comando "**para todo estado** α **faça ...**".

[11] Seria certamente mais simples assumir, por exemplo, que os estados finais são os *marcados*, mas essa estratégia não poderia ser levada a cabo sem perdermos a linearidade no tempo do algoritmo de construção.

Proposição 37 *A linha* 5 *é executada uma e apenas uma vez para cada estado do intervalo livre de* $\mathcal{A}$. *Ao término do laço* 4-7 *o valor de* α *é* μ *se esse estado estiver definido em* $\mathcal{A}$ *ou* ι *caso contrário.*

Prova: Segue diretamente da proposição 23, e da observação de que os atributos *pai* dos estados de $\mathcal{A}$ não são alterados pelas linhas 1-7.

Note que a linha 7 é executada se e somente se ι estiver no intervalo livre de $\mathcal{A}$, o que equivale a $s = \lambda$, e portanto o teste da linha 8 resulta verdadeiro se e somente se $s \neq \lambda$. Aplicando agora a proposição 22 provamos o

Corolário 38 *Se o estado* μ *estiver definido, então* **Passo** *inicializa a variável* μ *com o estado* μ *e a variável* θ *com o estado* θ.

Como as variáveis μ e θ armazenam os estados μ e θ, o teste realizado na linha 10 corresponde exatamente àquele da proposição 35, donde podemos concluir que o bloco 11-15 é executado se e somente se s for frouxa em x. Nossa atenção agora deve dirigir-se ao laço 12-14:

Proposição 39 *Se* s *for frouxa em* x *então a linha* 13 *é executada uma e apenas uma vez para cada estado do intervalo frouxo de* $\mathcal{A}$.

Prova: Como na entrada do laço temos $\alpha = \mu$, basta aplicar a proposição 24 observando, novamente, que nas linhas 1-14 não é alterado o atributo *pai* de nenhum estado de $\mathcal{A}$.

Nada nos impede de supor que o estado alocado incondicionalmente na linha 2 *seja* o estado ω', e que o estado alocado na linha 11 se s for frouxa em x *seja* o estado θ', donde podemos confundir as *variáveis* ω' e θ' com os *estados* homônimos.

Das proposições 37 e 39 segue que *a execução de* **Passo** *termina.* Isso autoriza-nos a falar no (ι, ω')*-autômato representado ao término da execução de* **Passo**. Esse autômato será denotado por $\mathcal{C}$. Já temos que $Q_{\mathcal{C}} = Q_{\mathcal{B}}$. Provaremos a seguir que $\mathcal{C} = \mathcal{B}$.

Proposição 40 $\mathcal{C}$ *e* $\mathcal{B}$ *têm as mesmas transições.*

Prova: Basta observar que as transições representadas pela estrutura de dados ao final do laço 4-7 são exatamente aquelas do conjunto sem redirecionamentos T_s (veja a proposição 30), e, quando s é frouxa em x, as transições

representadas pela estrutura de dados ao final do laço 12-14 são exatamente aquelas do conjunto com redirecionamentos T_c (proposição 30, novamente). Convém ressaltar que no momento em que a linha 11 é executada, as transições de θ que a estrutura de dados representa são exatamente as transições de θ em $\mathcal{B}$. De fato, da definição de intervalo frouxo vem que θ nunca pertence ao intervalo frouxo de $\mathcal{A}$, e portanto as transições de θ em $\mathcal{B}$ são as transições de θ em $\mathcal{A}$ mais (eventualmente) uma transição incluída. Como o laço 4-7 faz todas as inclusões necessárias, segue o que queríamos.

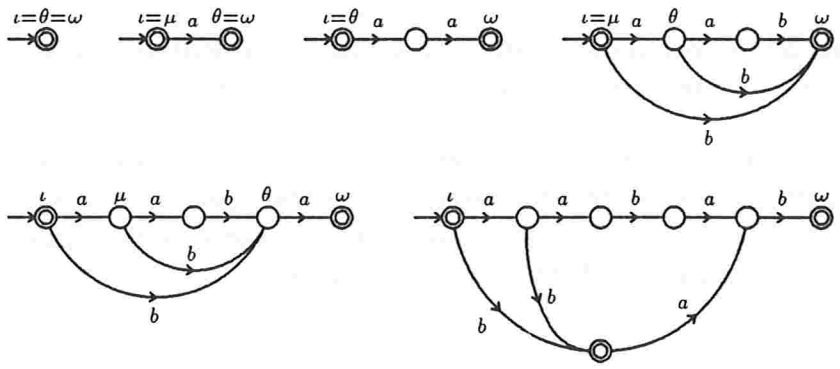


Figura 1.7: A construção incremental do Autômato dos Sufixos de *aabab*

Antes de concluir que $\mathcal{C} = \mathcal{B}$, é necessário verificar se $F_c = F_B$. Como a definição dos estados finais de $\mathcal{C}$ depende dos atributos *pai*, teremos que estudá-los agora.

Proposição 41 *Para cada estado α de $\mathcal{C}$, o atributo pai de α está definido se e somente se $\alpha \neq \iota$, e, no caso de estar definido,* pai[α] *é o pai de α em $\mathcal{B}$.*

Prova: Note que **Passo** define o atributo *pai* dos estados que cria (esses estados são, naturalmente, distintos de ι). Na linha 15 sabemos que $s \neq \lambda$, e portanto $\theta \neq \iota$. Como **Passo** só define (ou altera) o atributo *pai* de θ e dos estados que cria, a primeira parte do enunciado está provada. Quanto à segunda parte, basta seguir os enunciados das proposições 31, 32, 33 e 34, que determinam os pais em $\mathcal{B}$ de ω', θ e, quando θ' existir, de θ'.

Só nos resta provar que, para cada estado α de C, $prof[\alpha] = |y_\alpha|$. Após a execução da linha 2 temos que $prof[\omega'] = prof[\omega] + 1 = |x| + 1 = |y| = |y_{\omega'}|$. Por outro lado, se s for frouxa em x então após a execução da linha 11 temos que $prof[\theta'] = prof[\mu] + 1 = (|s| - 1) + 1 = |s| = |y_{\theta'}|$. Quanto aos outros estados, **Passo** não altera os seus atributos *profundidade*, e de $x_\alpha = y_\alpha$ para $\alpha \in Q_{\mathcal{A}}$ vem o que queremos.

1.4.3 Linearidade no tempo

Uma seqüência de $|x|$ chamadas da função **Passo** constrói o autômato dos sufixos de x, e para isso executa um total de operações [12] proporcional ao tamanho de x, independentemente do tamanho do alfabeto.

```
função AutSuf(x)
  NovoEstado(ι)
  prof[ι] ← 0
  ω ← ι
  para j de 1 até |x| faça ω ← Passo(ι, ω, x_j)
  enquanto ω estiver desmarcado faça
    marque(ω)
    se pai[ω] estiver definido então ω ← pai[ω]
  devolva(ι)
```

Algoritmo 1.2: A função AutSuf. O comando **enquanto** marca os estados finais de $\mathcal{A}$.

De fato, o total de vezes que a operação de alocação de estado é executada não pode ser maior do que o número de estados de $\mathcal{A}$ que, como sabemos, é menor ou igual a $2|x| - 1$ quando $|x| \geq 2$ (proposição 20). Um comentário análogo vale para a operação de criação de transição. Quanto aos redirecionamentos, o corolário 28 provou que $\sigma_y \leq |F_{\mathcal{A}}| - |F_B| + 2$. Assim, a soma

[12] Estamos entendendo por *operações* exatamente aquelas de que **AutSuf**, **Passo** e **Cópia** fazem uso direto, especialmente as operações sobre o autômato representado: alocação de estado, criação de transição, teste de existência de transição, obtenção ou alteração do seu destino, teste de definição, uso, definição e alteração de um atributo de um estado.

de todos os σ_u onde u percorre os prefixos de x (exceto λ), é telescópica e resulta não maior do que $1 - |F_A| + 2|x|$, expressão que por sua vez é menor ou igual a $2|x| - 1$ se $|x| > 0$. As outras operações não oferecem qualquer dificuldade. O número de vezes que a operação $\alpha \leftarrow pai[\alpha]$ (por exemplo) é realizada é claramente não superior à soma do total de transições criadas com o total de redirecionamentos.

Proposição 42 *O total de operações realizadas por* **AutSuf** *e pelas chamadas de* **Passo** *que ela realiza para construir o autômato dos sufixos de x é linear no comprimento de x, independentemente de $|A|$.*

Assim, se cada operação realizada por **AutSuf, Passo** e **Cópia** puder ser implementada em tempo constante, o tempo total consumido será linear no comprimento de x, independentemente do tamanho do alfabeto [13]. Isso pode ser conseguido se dispusermos de memória proporcional a $|x||A|$.

Procedimento Crie(α, b, β)
  $topo \leftarrow topo+1$
  $(lixo, c) \leftarrow t_\alpha[\perp]$
  $t_\alpha[b] \leftarrow (topo, c)$
  $t_\alpha[\perp] \leftarrow (lixo, b)$
  $T[topo] \leftarrow (\alpha, b, \beta)$

Função Destino(α, b)
  $(i, lixo) \leftarrow t_\alpha[b]$
  se $0 < i$ **e** $i \leq topo$ **então**
    $(\beta, c, \gamma) \leftarrow T[i]$
    se $\beta \neq \alpha$ **ou** $c \neq b$ **então** $\gamma \leftarrow \epsilon$
  devolva(γ)

Para cada estado α representado, aloca-se um vetor t_α de pares (n, b), onde n é um inteiro e b é um elemento de $A \cup \{\perp\}$ ($\perp \notin A$), com $|A| + 1$

[13] Isso subentende um modelo computacional teórico onde as operações envolvendo inteiros ou elementos de A (como a atribuição, por exemplo) são feitas em tempo constante. Na prática as coisas se passam de modo distinto: a leitura de um elemento de A, por exemplo, consome tempo $O(\log |A|)$.

entradas, indexadas por $A \cup \{\perp\}$ (essa segunda componente servirá para construir uma lista de todas as transições de cada estado). Podemos assumir que $t_\alpha[\perp]$ é inicializado com $(0, \perp)$. As demais entradas não serão, é claro, inicializadas, e por isso necessitaremos de um esquema de autenticações: T será um vetor com $3|x| - 2$ entradas indexadas pelos inteiros de 1 a $3|x| - 2$, cada uma delas uma ternas (α, b, β) onde α e β são estados e b é uma letra. Nessas condições, o procedimento **Crie** inclui a transição de α com b definindo o seu destino como sendo β (assumimos que *topo* vale zero inicialmente). É igualmente simples escrever um procedimento para redirecionamento de transições e uma função **Destino** (queira ver) que determina o destino da transição de um estado α com a letra b, retornando o valor especial ϵ se essa transição não existir.

Para concluir, note que a lista formada pelas segundas componentes do vetor t_α (o rótulo do primeiro elemento da lista é a segunda componente da entrada indexada por $\perp$) permite percorrer as transições de α em tempo proporcional ao número dessas transições, o que, entre outra coisas, viabiliza a implementação de **Cópia**.

1.4.4 Transformação do Autômato na Árvore

Podemos agora mostrar como a Árvore dos Sufixos de x^R pode ser obtida em tempo linear em $|x|$ independentemente do tamanho do alfabeto A a partir do autômato $\mathcal{A}$ e das estruturas de dados que acabamos de exibir.

Suponha que além dos atributos *pai* e *profundidade* a nossa estrutura de dados permita ainda para cada estado alocado α o atributo *primeiro término* $ptr[\alpha]$, que é um número inteiro. Ao alocar ω', **Passo** deverá definir o seu atributo *primeiro término* com o valor $prof[\omega']$, enquanto que o atributo *primeiro término* de θ' (se esse estado for criado) deverá ser definido com o valor $ptr[\theta]$.

É uma indução elementar a prova de que, ao término da construção de $\mathcal{A}$, o atributo *primeiro término* de cada estado $\alpha \neq \iota$ de $\mathcal{A}$ é, precisamente, o elemento mínimo de $t_x(x_\alpha)$ (em particular, para cada fator $u \in \mathrm{Fat}(x)$ esse atributo permite determinar através de $\mathcal{A}$ em tempo $O(|u|)$ a *primeira ocorrência* de u em x).

Para maior simplicidade na exposição, suponha que, em seguida à construção de $\mathcal{A}$, destruamos todas as suas transições, preservando apenas os estados e os seus atributos. Nesse ponto devemos supor que a palavra x está

armazenada na memória. Tomando então um estado $\alpha \neq \iota$ de $\mathcal{A}$, sejam $k = ptr[\alpha]$ e $\beta = pai[\alpha]$. Definindo $i = k - prof[\alpha] + 1$ e $j = k - prof[\beta] + 1$, é imediato que $x_i x_{i+1} \ldots x_k = x_\alpha$ e que $x_j x_{j+1} \ldots x_k = x_\beta$. Portanto o rótulo da transição de β para α em $\mathcal{T}^R$ é o reverso de $x_i x_{i+1} \ldots x_{j-1}$.

Assim, criemos uma transição de origem β, destino α e rótulo x_{j-1}. A letra x_{j-1} é na verdade apenas a *primeira* letra[14] do rótulo da transição de β para α em $\mathcal{T}^R$, e por isso será necessário associar a essa transição o par $(i, j-1)$ (na estrutura de dados que descrevemos não se previu esse gasto adicional, mas é fácil alterá-la para que se possa fazê-lo). A razão de associarmos às transições pares de índices ao invés dos próprios rótulos é que a soma dos comprimentos desses rótulos é, no pior caso, quadrática em $|x|$.

Proposição 43 *A Árvore dos Sufixos $\mathcal{T}^R$ do reverso de x pode ser construída em tempo linear no comprimento de x, independentemente do tamanho do alfabeto.*

Prova: Da proposição 16 vem que a repetição do procedimento que acabamos de descrever para cada estado $\alpha \neq \iota$ de $\mathcal{A}$ constrói $\mathcal{T}^R$, sendo necessário apenas fazer algum reparo em relação aos estados finais: supondo que a construção de $\mathcal{A}$ manteve todos os estados desmarcados e que antes de destruirmos as transições de $\mathcal{A}$ tenhamos marcado os estados de $\mathcal{A}$ que estão no caminho de $\mathcal{A}$ que soletra x a partir de ι, então os estados marcados serão exatamente os finais em $\mathcal{T}^R$.

Quanto ao tempo utilizado, a estrutura de dados que descrevemos permite destruir todas as transições de $\mathcal{A}$ em tempo linear em $|x|$ (basta fazer $topo \leftarrow 0$ e $t_\alpha[\perp] \leftarrow (0, \perp)$ para cada estado α). Para completar, basta observar que o total de transições criadas durante a construção de $\mathcal{T}^R$ é $|Q_\mathcal{A}| - 1 \leq 2|x| - 2$.

Tem alguma importância observar que, no autômato $\mathcal{A}$ construído por **AutSuf**, podemos percorrer todas as transições de um estado α em tempo proporcional ao número dessas transições (falamos disso anteriormente) *mas não na ordem alfabética dos seus rótulos*[15]. A possibilidade de se percorrer as transições de um estado na ordem alfabética dos seus rótulos é uma propriedade relevante de uma estrutura de dados. Ela permite-nos, por exemplo, ao representarmos um autômato acíclico $\mathcal{C}$, calcular eficientemente

[14] Vale lembrar que $\mathcal{T}^R$ é f-determinística.

[15] Por *ordem alfabética em A* entende-se qualquer ordem prefixada em A.

a posição (relativa à ordem lexicográfica) de uma palavra dada na linguagem reconhecida por $\mathcal{C}$, o que é útil na implementação de índices baseados em autômatos.

Na construção da árvore $\mathcal{T}^R$ que descrevemos isso pode ser parcialmente remediado, no seguinte sentido: é possível, mantendo a linearidade no tempo, obter $\mathcal{T}^R$ com as listas das transições de cada estado ordenadas na ordem lexicográfica dos seus rótulos, desde que troquemos a ordem alfabética pela *ordem de aparecimento em* x dessas letras: se a e b ocorrem em x, então a precede b na ordem de aparecimento em x se e somente se $i < j$, onde i e j são os menores índices em $\{1, 2, \ldots, |x|\}$ que satisfazem $x_i = a$ e $x_j = b$. Sem esse relaxamento, o problema é insolúvel, como veremos no capítulo 2. É lá, de fato, que essa construção ganha sentido, pois ela nos levará a um algoritmo tempo-linear para a construção do Vetor dos Sufixos.

Vejamos como fazê-lo. Será necessário dividir os estados de $\mathcal{A}$ (exceto ι), em listas de acordo com as primeiras letras dos rótulos das transições que os atingem em $\mathcal{T}^R$. Sejam $a_1, a_2, \ldots, a_m$ as letras de x dispostas na ordem de aparecimento em x. Considere os vetores v e w indexados por $\{1, 2, \ldots, m\}$ e A respectivamente, e definidos por $v[i] = a_i$ e $w[a_i] = i$ (Para que não precisemos inicializar w, se $a \in A \setminus \{a_1, a_2, \ldots, a_m\}$, a entrada $w[a]$ deve ser considerada *indefinida*). Os vetores v e w podem ser trivialmente obtidos ao longo da construção de $\mathcal{A}$ e o tempo total gasto para isso é linear em $|x|$ independentemente do tamanho do alfabeto (a memória é naturalmente $O(|x| + |A|)$).

Construído $\mathcal{A}$, procedamos à divisão dos seus estados nas listas de que falamos: para cada estado $\alpha \in Q_{\mathcal{A}} \setminus \{\iota\}$, determine a primeira letra b do rótulo da transição de $\mathcal{T}^R$ que atinge α (já indicamos como obter essa letra), e ponha α na lista $L_{w[b]}$. Note que $m \leq |x|$ e portanto podemos inicializar as listas $L_1, L_2, \ldots, L_m$ sem com isso perder a linearidade em $|x|$ do tempo total.

Passemos agora à construção de $\mathcal{T}^R$ propriamente dita. Faz-se exatamente como descrevemos antes, com a diferença de que deve-se visitar inicialmente os estados de L_1, em seguida os de L_2, e assim por diante até L_m. Com isso garante-se que se α tem em $\mathcal{T}^R$ as transições t_b e t_c cujos rótulos têm como primeiras letras respectivamente b e c, então $w[b] < w[c]$ se e somente se t_b tiver sido criada antes de t_c, e portanto as listas das transições na nossa estrutura de dados estão na ordem inversa do aparecimento em x. Se o procedimento **Crie** incluísse a transição recém-criada no *final* da lista, então

elas estariam ordenadas segundo a ordem do aparecimento em x.

1.5 Representação

O esquema de autenticações dado em 1.4.3 para a representação do autômato dos sufixos é apenas um artifício para mostrar que o tempo total da sua construção pode independer do tamanho do alfabeto. Na prática o fator crítico para essa construção é o consumo de memória, e não podemos darnos ao luxo de usar estratégias dispendiosas como aquela.

Pode-se remover o fator $|A|$ do consumo de memória colocando-o no tempo de execução. Para isso, basta representar as transições de cada estado por uma lista linear ([4] faz assim, com a ressalva de que lá o tamanho do alfabeto é considerado limitado por uma constante, donde o tempo torna-se também linear em $|x|$). Se ao invés de listas usarmos árvores balanceadas, o fator que surge no tempo de execução é $\log |A|$ ([7] fala nessa possibilidade).

Não se conhece atualmente uma estratégia que permita representar o autômato dos sufixos de tal forma que o consumo de memória e o tempo total consumido pelo algoritmo de construção sejam lineares em $|x|$, independentemente do tamanho do alfabeto. No que segue exibiremos dois resultados negativos, que podem eventualmente ser úteis para estudos futuros. Trata-se de duas estratégias para a representação do autômato dos sufixos. A primeira permite a implementação das operações de que necessitamos em tempo constante. A segunda não sabemos se permite ou não. Para as duas, entretanto, daremos contra-exemplos para a afirmação de que o consumo de memória é linear em $|x|$.

1.5.1 Distribuições concentradas

O número $q_x = q_x(k)$ de estados de $\mathcal{A}$ com pelo menos k saídas, onde $k > 0$ é um inteiro, é limitado superiormente por $3|x|/k$, visto que $3|x|$ majora o total de transições de $\mathcal{A}$.

Considere a seguinte estratégia para a representação do autômato dos sufixos de x: associaremos a cada estado com pelo menos k saídas um vetor de $|A|$ entradas, sendo que a entrada indexada por b será o destino da transição desse estado com b, ou um valor nulo no caso dessa transição não existir (subentende-se que esse vetor deverá ser inicializado). As transições de um

estado com menos de k saídas serão representadas por uma lista linear. Se existir um valor de k tal que, ao fazermos $|A|$ e x variar, o produto $q_x|A|$ se mantenha sempre inferior a $m|x|$, onde m é uma constante independente de A e de x, então para esse valor de k essa estratégia resolve o problema da representação do autômato dos sufixos em espaço ótimo que permite a implementação em tempo ótimo das operações sobre autômatos de que **AutSuf, Passo** e **Cópia** necessitam. Provaremos, entretanto, a inexistência de um valor de k com essa propriedade.

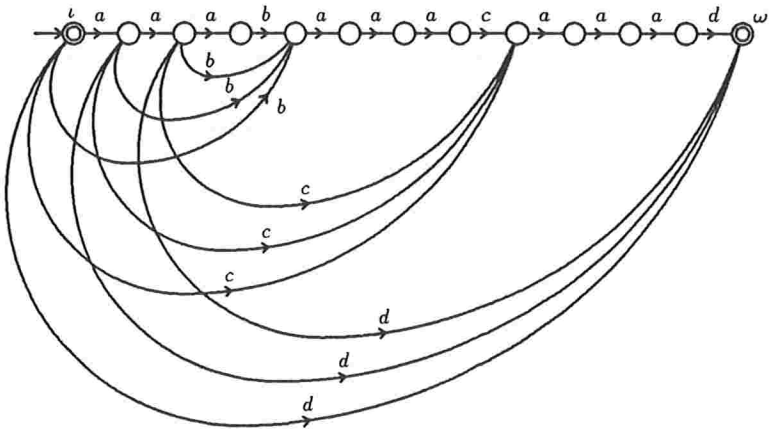


Figura 1.8: o autômato dos sufixos de $a^3ba^3ca^3d$.

Proposição 44 *Fixado $k > 0$, para cada alfabeto finito A com $|A| \geq k$, existe uma linguagem infinita $L = L(A,k)$ tal que para cada $u \in L$, o produto $q_u|A|$ supera $|u||A|/k$.*

Prova: Podemos supor $k \geq 2$. Sejam $a_1, a_2, \ldots, a_{|A|} = a$ as letras de A. Considere a palavra $u = u(r) = a^r a_1 a^r a_2 \ldots a^r a_{k-1}$. Seja $\mathcal{U} = \mathrm{AutSuf}(u)$ e α o estado inicial de $\mathcal{U}$. Para cada $i \in \{0, 1, \ldots, r-1\}$ a palavra a^i é prefixo e, portanto, fator esticado de u. Assim o conjunto $T(u) = \{(\alpha a^i)_{\mathcal{U}}; 0 \leq i < r\}$ tem r elementos, e cada um deles tem exatamente k saídas (pois $a^i b$ é fator de u se (e somente se) $b \in \{a, a_1, a_2, \ldots, a_{k-1}\}$). Supondo $r \geq k$, de $|u| = (k-1)(r+1)$ vem $r \geq |u|/k$, e basta colocar $L = \{u(r); r \geq 1\}$.

O exemplo dado na proposição anterior é fraco em dois sentidos. O primeiro é que o número de letras distintas que ocorrem em u é exatamente

k, e portanto o produto de q_u pelo tamanho do alfabeto *realmente* utilizado é $|u|$. Essa objeção entretanto pode ser facilmente rebatida: pondo $v = v(r) = a^r v_1 a^r v_2 \ldots a^r v_{k-1}$ onde $r \geq 1$ e v_i é uma palavra em $a_i(A \setminus \{a\})^*$ de comprimento r (por exemplo), temos que $q_v \geq q_u$ e que $|v| < 2|u|$, donde $q_v|A| \geq |v||A|/(2k)$.

O segundo sentido em que o exemplo é fraco é que é desnecessário alocar um vetor para cada estado de $T(u)$, pois dois desses vetores diferem apenas na entrada indexada por a. Isso significa que ao invés de alocarmos r vetores, podemos alocar um único, através do qual não só decidiremos, para cada estado β de $T(u)$ e para cada letra b de $A \setminus \{a\}$, se β tem a transição com b, como também, em caso afirmativo, determinaremos o destino dessa transição. Naturalmente, cada um desses estados deverá contar com espaço adicional para armazenar o destino da sua transição com a. A generalização dessa idéia leva-nos a uma estratégia de representação que apresentaremos a seguir.

1.5.2 Cobertura módulo k

Dados $\alpha, \beta \in Q_{\mathcal{A}}$, diremos que α e β *diferem em relação à letra* $b \in A$ se um e apenas um dos estados α, β tiver em $\mathcal{A}$ a transição com b ou se os dois a tiverem e os destinos delas forem distintos. Definimos a *diferença* (ou *distância de Hamming*) $\Delta_{\mathcal{A}}(\alpha, \beta)$ como sendo o número de letras $b \in A$ em relação às quais α e β diferem. Seja $k > 0$ um inteiro. Uma *cobertura módulo* k de $\mathcal{A}$ é um subconjunto $K = K(k, \mathcal{A}) \subseteq Q_{\mathcal{A}}$ tal que para todo estado $\alpha \in Q_{\mathcal{A}}$ existe um estado $\beta = \beta(\alpha)$ em K que satisfaz $\Delta_{\mathcal{A}}(\alpha, \beta) < k$.

Proposição 45 *Existe uma cobertura módulo 2 de* $\mathcal{U} = \mathrm{AutSuf}(u)$ *com apenas dois elementos, onde* $u = u(r)$ *é a palavra definida na demonstração da proposição 44 e* r *é arbitrário.*

Prova: Sejam α um elemento de $T(u)$ (essa notação foi definida na demonstração da proposição 44) e β o sorvedouro de $\mathcal{U}$. Provaremos que $\{\alpha, \beta\}$ é uma cobertura módulo 2 de $\mathcal{U}$. Tome $\gamma \in Q_{\mathcal{U}}$ distinto de α e de β. Se $\gamma \in T(u)$, então já provamos que $\Delta_{\mathcal{U}}(\alpha, \beta) = 1$. Se $\gamma \notin T(u)$, então uma rápida inspeção mostra que ou $u_\gamma = a^r$ (caso em que $\Delta_{\mathcal{U}}(\alpha, \gamma) = 1$) ou alguma letra de $\{a_1, a_2, \ldots, a_{k-1}\}$ ocorre em u_γ, donde u_γ ocorre em u uma única vez e portanto γ tem exatamente uma saída e segue $\Delta_{\mathcal{U}}(\beta, \gamma) = 1$.

Uma cobertura módulo k de $\mathcal{A}$ permite representar $\mathcal{A}$ através da seguinte estratégia: as transições dos estados da cobertura são representadas por um

vetor de $|A|$ entradas, e, as transições de um estado α fora da cobertura, por um apontador para um estado β na cobertura com $\Delta_{\mathcal{A}}(\alpha, \beta) < k$, mais uma lista das (no máximo) $k-1$ *diferenças*. Assim, para toda letra b de A, podemos decidir em tempo $O(k)$ se α tem a transição com b, e, em caso afirmativo, determinar o seu destino.

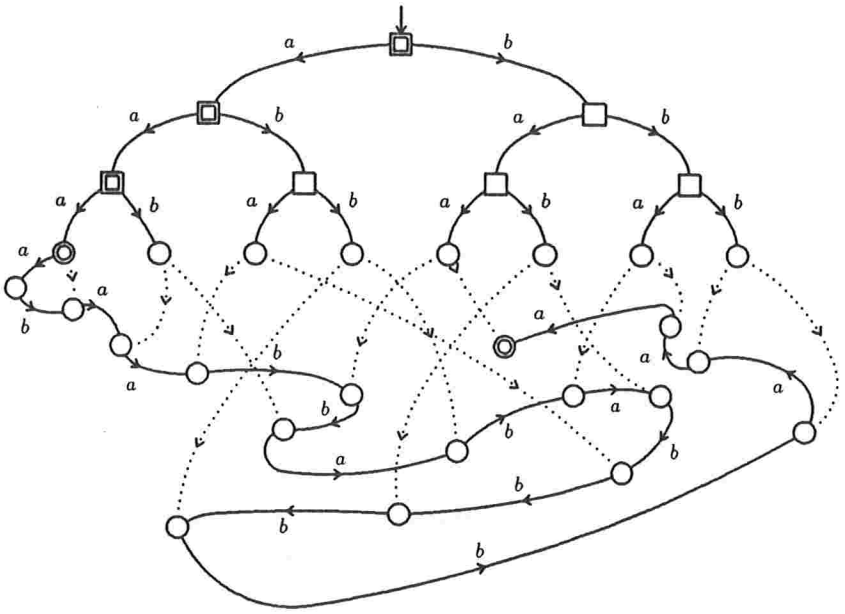


Figura 1.9: A Árvore esticada do autômato dos sufixos de *aaaabaabbababbbbaaa*. Essa palavra é de Good-de Bruijn sobre $\{a, b\}$ a parâmetro 4. Os estados quadrados são aqueles que na proposição 46 demonstra-se pertencerem a qualquer cobertura módulo 2. As transições não esticadas estão pontilhadas.

Seja $p_x = p_x(k)$ o tamanho de uma cobertura módulo k mínima de $\mathcal{A}$. Analogamente à estratégia anterior, se existir um valor de k tal que, ao fazermos A e x variar, o produto $p_x|A|$ se mantenha sempre inferior a $m|x|$, onde m é uma constante independente de A e de x, então para esse valor de k essa estratégia oferece uma representação do autômato dos sufixos em espaço ótimo. Quanto à implementação das operações sobre autômatos de que **AutSuf**, **Passo** e **Cópia** necessitam, não há muito que possamos dizer. O teste da existência da transição com a de α pode naturalmente ser imple-

mentado em tempo constante, assim como o cálculo do seu destino, quando essa transição existir. Entretanto, ao criarmos uma transição, a cobertura de que dispúnhamos pode deixar de ser cobertura, ou, ainda que continue sendo, pode não ser mais mínima. De fato, sequer sabemos calcular p_x eficientemente. O que podemos provar, e que de certa forma já é suficiente, é que um valor de k com a propriedade de que necessitamos não existe. É claro, entretanto, que isso não prova a inexistência de uma estratégia para a representação do autômato dos sufixos ótima em tempo e espaço.

Dado um inteiro $r > 0$, x é uma *palavra de Good-de Bruijn sobre A a parâmetro r* se $A^r \subseteq \mathrm{Fat}(x)$ e $|x| = |A|^r + r - 1$. A construção clássica que prova a existência de palavras de Good-de Bruijn é a seguinte: seja $G = G(A, r)$ o grafo dirigido com conjunto de vértices A^{r-1} onde existe uma aresta de u para v rotulada com b se e somente se v for sufixo de ub. Para cada estado de G, o grau de entrada é igual ao grau de saída (ambos valem $|A|$), e portanto existe em G uma trilha orientada e fechada de Euler. Seja $t_1, t_2, \ldots, t_n$ a seqüência de arestas de uma tal trilha. Então $n = |A|^r$ e $a_1 a_2 \ldots a_n a_1 a_2 \ldots a_{r-1}$ onde a_i é o rótulo de t_i é de Good-de Bruijn sobre A a parâmetro r.

Não é difícil descrever completamente o autômato dos sufixos $\mathcal{A}$ quando x é de Good-de Bruijn sobre A a parâmetro r. Vamos verificar, por exemplo, que, supondo $|A| > 1$, o número de estados de $\mathcal{A}$ é exatamente $(|A|^{r+1} - 1)/(|A| - 1)$. Note que $v \in \mathrm{Fat}(x)$ satisfaz $|v| < r$ se e somente se para duas letras quaisquer $b, c \in A$ valer $av, bv \in \mathrm{Fat}(x)$. Daí segue que v é esticada em x se e somente se v for prefixo de x ou se $|v| < r$, e basta agora fazer uma contagem elementar.

Proposição 46 *Fixado $k > 1$, para cada alfabeto finito A com $|A| \geq k$, existe uma linguagem infinita $L = L(A, k)$ tal que para cada $u \in L$, o produto $p_u(k)|A|$ supera $|u|/(2k^2)$.*

Prova: Seja B um subconjunto de A com k elementos. Para cada $r > 0$, ponha $u = u(r) = v_r w_r$ e $\mathcal{U} = \mathrm{AutSuf}(u)$ onde v_r é de Good-de Bruijn sobre B a parâmetro r e w_r é uma palavra qualquer sobre A de comprimento, digamos, menor ou igual a $k^r - r + 1$ (o papel de w_r é apenas evitar uma objeção análoga à primeira feita ao exemplo dado na demonstração da proposição 44). Sejam z uma palavra sobre B de comprimento menor ou igual a $r - 2$ e α o destino do caminho que soletra u em $\mathcal{U}$ a partir do estado inicial. Vamos provar que α pertence a qualquer cobertura módulo k de $\mathcal{U}$.

Sejam b uma letra de B e β um estado de $\mathcal{U}$ que tem a transição com b. Sejam γ e δ os destinos das transições de α e β com b em $\mathcal{U}$. Então a igualdade $\gamma = \delta$ acarreta $\alpha = \beta$. De fato, zb é esticada em u, e portanto $u_\beta b$ é sufixo de zb, mas não próprio, pois todo sufixo próprio de zb é fator esticado de u.

Daí segue que se $\beta \in Q_\mathcal{U} \setminus \{\alpha\}$, então α e β diferem em relação a qualquer letra $b \in B$, donde $\Delta_\mathcal{U}(\alpha, \beta) \geq k$, que é o que queríamos. Ora, o número de palavras z sobre B com comprimento menor ou igual a $r-2$ é exatamente $(k^{r-1}-1)/(k-1)$, e portanto $p_u(k) \geq k^{r-2}$, donde segue por cálculos simples que $p_u(k)|A| \geq |u|/(2k^2)$. Assim, $L = \{u(r); r \geq 1\}$ satisfaz o enunciado.

Capítulo 2

O Vetor dos Sufixos

Trata-se de uma estrutura de dados bastante simples, que conta com duas recentes aplicações, a mais significativa delas no projeto de informatização do Dicionário de Oxford. A origem do vetor dos sufixos está fortemente ligada às aplicações de computação em biologia molecular. De fato, a primeira referência a ele que conhecemos encontra-se num método [20] para a determinação de repetições em seqüências biológicas (falaremos delas no capítulo 3), além do que as aplicações em biologia estão sempre presentes entre as motivações dos pesquisadores que trataram dessa estrutura.

Não há um critério absoluto que permita comparar o vetor dos sufixos com as outras duas estruturas baseadas em sufixos que apresentamos no capítulo 1, mas pode-se dizer que a vantagem fundamental do vetor sobre o autômato e a árvore dos sufixos é a economia de memória, suficiente por vezes para viabilizar um sistema onde o uso do autômato ou da árvore são proibitivos.

O vetor dos sufixos da palavra x pode ser obtido como subproduto da construção da árvore dos sufixos de x em tempo linear em $|x|$. Naturalmente não é assim que se constrói o vetor dos sufixos na prática, visto que a intenção ao optar-se por ele é justamente a economia de memória, e por isso é necessário lançar-se mão de métodos não tão eficientes com relação ao tempo. Apresentaremos dois algoritmos para a construção do vetor dos sufixos, um bastante elegante, para o qual desenvolvemos uma implementação econômica, e outro eminentemente prático, baseado em técnicas de ordenação externa, que foi desenvolvido durante a já referida informatização do Dicionário de Oxford.

2.1 Propriedades Básicas

A aplicação fundamental do vetor dos sufixos é o problema da busca do padrão u num texto x, que ele permite resolver em tempo $O(|u| \log |x|)$. Nesse ponto o vetor dos sufixos é complementar ao autômato dos sufixos, no sentido em que com o vetor resolve-se de forma muito natural o problema de determinar *todas* as ocorrências de u em x, mas não o de determinar a *primeira* ocorrência de u em x, enquanto no autômato[1] resolve-se de forma natural o problema de determinar a *primeira* ocorrência de u em x, mas não o de determinar todas elas. Veremos ainda que o vetor dos sufixos não permite simular eficientemente a árvore dos sufixos.

2.1.1 Busca de Padrões

Neste capítulo será conveniente descartar o sufixo λ, e por isso introduziremos a notação $S = S(x) = \mathrm{Suf}(x) \setminus \{\lambda\}$. Seja $f_x : \{1, 2, \ldots, |x|\} \to S$ a função injetora que satisfaz $f_x(i) \preceq f_x(j)$ se $1 \leq i < j \leq |x|$, onde $\preceq$ é a ordem lexicográfica em A^*.

A estrutura de dados natural para a implementação de f_x é um vetor f de inteiros indexado por $\{1, 2, \ldots, |x|\}$. A i-ésima entrada de f conterá, evidentemente, o índice j dado por $f_x(i) = x_j x_{j+1} \ldots x_{|x|}$. Esse vetor f é o *Vetor dos Sufixos* de x.

Dispondo-se de x e do vetor dos sufixos de x, pode-se decidir, através de uma busca binária, se $u \in A^*$ é ou não fator de x. Mais ainda, dado $u \in \mathrm{Fat}(x)$, duas buscas binárias bastam para determinar o *par característico* $(i, j) \in \{1, 2, \ldots, |x|\}^2$ de u (em x) definido por "$i \leq k \leq j$ se e somente se u for prefixo de $f_x(k)$". Isso equivale, naturalmente, a determinar *todas* as ocorrências de u em x. Os dois problemas (decidir se $u \in \mathrm{Fat}(x)$ e determinar todas as ocorrências de $u \in \mathrm{Fat}(x)$ em x) admitem portanto através do vetor dos sufixos algoritmos que os resolvem em tempo $O(|u| \log |x|)$. Em [19] eles são acelerados para $O(|u| + \log |x|)$, ao custo de $2|x|$ variáveis inteiras adicionais.

Deve-se ressaltar que, se (i, j) for o par característico de u em x, então a seqüência $f[i], f[i+1], \ldots, f[j]$ não é em geral ordenada. Em particular, o par (i, j) não nos dá de imediato a *primeira* ocorrência de u em x. A

[1] Poderia-se dizer o mesmo em relação à árvore dos sufixos.

determinação dessa primeira ocorrência a partir de (i, j) consome tempo proporcional ao número de ocorrências de u em x, visto que exige o cálculo do elemento mínimo do conjunto $\{f[i], f[i+1], \ldots, f[j]\}$.

(1)	9	ab
(2)	1	abababbbab
(3)	3	ababbbab
(4)	5	abbbab
(5)	10	b
(6)	8	bab
(7)	2	bababbbab
(8)	4	babbbab
(9)	7	bbab
(10)	6	bbbab

Figura 2.1: O vetor dos sufixos de $x = ababababbbab$ e os sufixos de x na ordem lexicográfica. O par característico de ab é $(1, 4)$ e o de bab é $(6, 8)$. Nas nossas figuras índices de vetores serão sempre dados entre parênteses.

A busca de padrões é o recurso básico que o sistema desenvolvido durante a informatização do *Dicionário de Oxford* oferece. O texto desse dicionário tem, com alguns acréscimos, cerca de 570 megabytes [2]. Esse sistema é baseado no vetor dos sufixos, mas nele não se considerou *todos* os sufixos do texto, como veremos em 2.3.

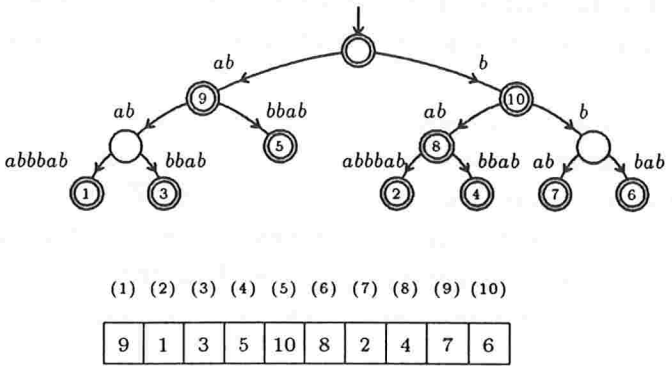
O sistema é interativo; a cada passo define-se uma coleção de *posições* do texto como resultado de uma busca. Pode-se especificar *padrões simples* ("war and peace"), *uniões* ("man" **or** "men") ou *intervalos* ("1769" .. "1775"). O resultado de duas ou mais buscas pode ser combinado através de vários comandos, entre eles os de *busca por proximidade*, que selecionam, dadas duas coleções de posições, os pares (uma componente de cada conjunto) formados por posições *próximas* entre si. Com esses recursos pode-se responder a diversas questões potencialmente interessantes para um literato ou um filólogo [11]. O algoritmo que foi utilizado para construir o vetor dos sufixos foi descrito em [12] e será apresentado em 2.3.

[2] Nós estimamos o texto do dicionário *Aurélio* em 24 megabytes. O Dicionário de Oxford contém milhões de citações que mostram a evolução de cada palavra da língua inglesa ao longo dos séculos, daí o seu tamanho avantajado.

2.1.2 Construção em tempo linear

Vejamos como o vetor dos sufixos de x pode ser obtido como subproduto da construção da árvore dos sufixos de x. Para que tenhamos um tempo de construção independente do tamanho do alfabeto, será necessário abrir mão da ordem alfabética em A trocando-a pela ordem do aparecimento em x^R (o vetor assim obtido permitiria normalmente buscas de padrões simples, mas não buscas de intervalos como "1769" .. "1775"). De fato, como a construção do Vetor dos Sufixos resolve em particular o problema da ordenação de um vetor de ítens, não se pode consegui-la em tempo linear sem hipóteses adicionais.

Já vimos que podemos obter em tempo linear em $|x|$ o autômato dos sufixos de x^R e, em seguida, a árvore dos sufixos $\mathcal{T}$ de x com as transições de cada estado dispostas numa lista ordenada segundo as primeiras letras dos rótulos de acordo com a ordem do aparecimento em x^R. Portanto podemos visitar os estados de $\mathcal{T}$ na ordem lexicográfica (relativa à ordem de aparecimento) das respectivas palavras realizadas consumindo para isso tempo $O(|x|)$ e, eventualmente, espaço $O(|x|)$. Isso leva-nos imediatamente ao vetor dos sufixos: tome um contador i inicialmente zerado; ao visitarmos $\alpha \neq \iota$, se α for final em $\mathcal{T}$ então incrementamos i de 1 e definimos $f[i]$ como sendo $|x| - ptr[\alpha] + 1$. Ao final, temos que $i = |x|$ e que f é o vetor dos sufixos de x.



(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
9	1	3	5	10	8	2	4	7	6

Figura 2.2: A árvore e o vetor dos sufixos de $x = ababababbbab$. Cada estado final α da árvore (exceto a raiz) está rotulado com $|x| - ptr[\alpha] + 1$.

De fato, sejam $\alpha \neq \iota$ um estado final de $\mathcal{T}$ e u a palavra realizada por α

em $\mathcal{T}$. Então $ptr[\alpha]$ é o menor término de u^R em x^R. Como u^R é prefixo de x^R, segue que $ptr[\alpha] = |u|$, e portanto colocando $j = |x| - ptr[\alpha] + 1$ vem que $u = x_j x_{j+1} \ldots x_{|x|}$, e como por hipótese os estados de $\mathcal{T}$ são visitados na ordem lexicográfica das respectivas palavras realizadas, está demonstrado o que queríamos.

2.1.3 O Vetor e a Árvore

O vetor dos sufixos foi apresentado como uma alternativa econômica para a árvore dos sufixos [19]. Ainda que problemas solúveis eficientemente através da árvore dos sufixos possam também sê-lo via vetor dos sufixos (um exemplo trivial disso é decidir se $y \in \mathrm{Fat}(x)$), não se pode dizer que o vetor dos sufixos *substitua* a árvore dos sufixos, pois, ao menos até onde sabemos, a simulação desta por aquele é cara.

A operação fundamental que a árvore dos sufixos $\mathcal{T}$ de x permite é, dado um estado α de $\mathcal{T}$ e uma letra $a \in A$, determinar se α possui ou não uma transição cujo rótulo u tem a como primeira letra e, em caso afirmativo, obter o seu rótulo e o seu destino. Não é difícil (já vimos como fazê-lo) representar $\mathcal{T}$, de tal forma que se possa implementar essa operação em tempo constante ou proporcional ao (logaritmo do) número de transições de α em $\mathcal{T}$. A simulação dessa operação através do vetor dos sufixos, entretanto, consome tempo proporcional ao comprimento de u. Na verdade poderíamos dizer que *a árvore que o vetor dos sufixos permite simular é a árvore digital* de $\mathrm{Suf}(x)$ (que pode ter número de estados quadrático em $|x|$), e não a árvore dos sufixos de x.

Vejamos isso com mais cuidado. A cada estado α de $\mathcal{T}$ associa-se naturalmente o seu *par característico* $(i, j) \in \{1, 2, \ldots, |x|\}^2$ que, por definição, é o par característico da palavra u realizada por α em $\mathcal{T}$. Se dispusermos de (i, j) e de u então podemos determinar através de x e do vetor dos sufixos de x em tempo $O(\log |x|)$ se, dada uma letra $a \in A$, α tem ou não em $\mathcal{T}$ uma transição com rótulo começando com a. De fato, isso equivale a decidir se ua é ou não fator de x.

Mais do que isso, pode-se determinar, sem acréscimo algum no tempo consumido, o par característico do destino dessa transição, já que isso equivale, como veremos na proposição que segue, a determinar o par característico de ua.

Proposição 47 *Seja u a palavra realizada pelo estado α de $\mathcal{T}$. Suponha que α tenha em $\mathcal{T}$ uma transição com rótulo v começando com a, e seja β o destino dessa transição. Então o par característico de β é o par característico de ua.*

Prova: Seja z sufixo de x, e suponha que ua seja prefixo de z. O estado β pertence necessariamente ao caminho de $\mathcal{T}$ que soletra z, e portanto uv é prefixo de z. Assim para todo $z \in \mathrm{Suf}(x)$ temos que ua é prefixo de z se e somente se uv for prefixo de z, donde segue o enunciado.

Ainda que possamos (usando a notação da proposição 47) obter eficientemente o par característico de β, o mesmo não se pode dizer do rótulo v. E, sem v, não podemos, por exemplo, prosseguir a simulação da árvore a partir dos filhos de α. A determinação de v equivale à obtenção do maior prefixo comum a duas palavras, que exige tempo proporcional ao comprimento desse maior prefixo comum, que no nosso caso é v.

Proposição 48 *Seja (i, j) o par característico do estado α de $\mathcal{T}$. Se α não for a raiz, então a palavra u realizada por α em $\mathcal{T}$ é o maior prefixo comum a $f_x(i)$ e $f_x(j)$.*

Prova: Sabemos que u ou é sufixo de x ou pode ser estendido para dois fatores $ua \neq ub$ de x $(a, b \in A)$. No primeiro caso é imediato que $f_x(i) = u$. Supondo agora que u não seja sufixo de x, sejam a (respectivamente b) a primeira (respectivamente última) letra na ordem alfabética que satisfaz $ua \in \mathrm{Fat}(x)$ (respectivamente $ub \in \mathrm{Fat}(x)$). Então ua é prefixo de $f(i)$ e ub é prefixo de $f(j)$, donde está provada a proposição.

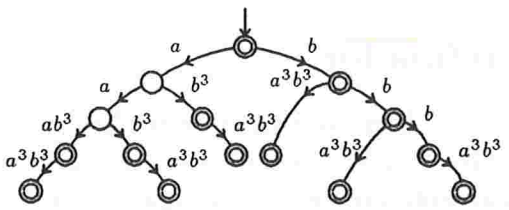


Figura 2.3: A árvore dos sufixos de $(a^3b^3)^2$.

Assim, suponha que desejamos percorrer todos os estados de $\mathcal{T}$ simulando-a através do vetor dos sufixos de x. A aplicação da estratégia que descrevemos executaria no mínimo um número de comparações de caracteres igual à soma dos comprimentos dos rótulos de $\mathcal{T}$ que não atingem folhas de $\mathcal{T}$. Ora, se $x = (a^r b^r)^2$ onde a e b são letras distintas, essa soma é $r^2 + 2r$, e portanto quadrática no comprimento de x.

Apesar disso essa simulação foi levada a cabo no programa *Todos-contra-todos* [2]. As seqüências de um banco de proteínas foram concatenadas dando origem a uma palavra x de comprimento um milhão aproximadamente. Em seguida determinou-se a *similaridade* (veja 3.1.2) entre cada dois fatores de seqüências do banco. A simulação da árvore dos sufixos permite alguma economia de tempo por evitar repetição de trabalho. Aliando a isso algumas otimizações e vários computadores trabalhando simultaneamente, pôde-se realizar todos os cálculos em tempo hábil (cerca de quatro meses, ou o equivalente a um ano de processamento de uma estação de trabalho). Um algoritmo ingênuo nesse caso consumiria tempo proporcional a $|x|^4$ (veja 3.1.2), proibitivo em vista do tamanho do banco.

2.2 Um algoritmo de refinamento

Desenvolveremos a apresentação do algoritmo de refinamento de Manber e Myers de forma abstrata, para em seguida dar uma implementação dele que utiliza memória equivalente a $2|x|$ variáveis inteiras e tempo total $O(|x| \log |x|)$. A existência de uma tal implementação é afirmada em [19], mas não descrita. O consumo de memória desse algoritmo é significativamente menor que o daquele descrito em 2.1.2, mas a sua aplicação prática é limitada principalmente pela virtual impossibilidade de se usar memória secundária. Veremos em 2.3 um algoritmo que não possui essa deficiência.

2.2.1 O passo refinador

O algoritmo de Manber e Myers obtém o vetor dos sufixos de x através de seqüência de *refinamentos*. Cada passo consiste em, a partir dos sufixos de x ordenados lexicograficamente em relação às suas k primeiras letras, ordená-los lexicograficamente em relação às suas $2k$ primeiras letras. Assim, partindo dos elementos de S ordenados segundo as suas primeiras letras, constrói-se

o vetor dos sufixos de x com $\lceil \log_2 |u| \rceil$ chamadas do passo refinador onde u é o comprimento do fator repetido mais longo de x.

Precisaremos de algumas notações. Para cada $u \in A^*$ denotaremos por $u(i,j)$ a palavra $u_i u_{i+1} \ldots u_j$ (se $i \leq j \leq |u|$) ou a palavra $u_i u_{i+1} \ldots u_{|u|}$ (se $i \leq |u| < j$), ou a palavra λ (se $|u| < i$). Se $k > 0$ e $R \in \{=, \prec, \preceq\}$, escreveremos $u\ R_k\ v$ para significar $u(1,k)\ R\ v(1,k)$, e $u\ R^k\ v$ para significar $u(k+1,2k)\ R\ v(k+1,2k)$.

Naturalmente $=_k$ e $=^k$ são relações de equivalência em S. Quando C, C' forem classes de S sob $=_k$ ($=^k$) escreveremos $C \preceq_k C'$ ($C \preceq^k C'$) se valer $u \preceq_k v$ ($u \preceq^k v$) para $u \in C, v \in C'$ arbitrários. Com essas definições, $\preceq_k$ ($\preceq^k$) é uma ordem total no conjunto das classes de S sob $=_k$ ($=^k$), e quando falarmos na *i-ésima* classe de S sob $=_k$ ($=^k$) estaremos subentendendo $\preceq_k$ ($\preceq^k$). Dado $u \in S$, denotaremos por $\overline{u}$ a classe de S sob $=_k$ que contém u (não necessitaremos de uma notação para a classe de S sob $=^k$ que contém u, e por isso não tem grande importância que a notação $\overline{u}$ não explicite a relação $=_k$).

Para cada $u = x_i x_{i+1} \ldots x_{|x|} \in S$ com $i > k$ definimos o *k-precursor* de u (omitiremos k) como sendo o sufixo $x_{i-k} x_{i-k+1} \ldots x_{|x|}$ (o precursor está indefinido para os demais elementos de S). Diremos que um subconjunto de S sob $=_k$ *tem precursores* se algum elemento seu tiver precursor, e o conjunto de tais precursores será chamado de *conjunto dos precursores* desse subconjunto.

Admitiremos que a nossa estrutura de dados representa classes através de listas não necessariamente ordenadas dos seus elementos, e também que as classes disponham-se numa lista. Ao representar as classes de S sob $=_k$, a lista das classes deverá estar ordenada segundo $\preceq_k$. Nessas condições, podemos realizar a seguinte operação fundamental: visitar na ordem $\preceq_k$ as classes de S sob $=_k$ e, para cada classe visitada, visitar todos os seus elementos (na ordem em que eles estiverem dispostos na lista que representa essa classe). Dessa operação poderemos derivar uma outra, mas antes precisamos de uma proposição.

Proposição 49 *A primeira classe de S sob $=^k$ é $\{u \in S; |u| \leq k\}$. A j-ésima ($j \geq 2$) classe de S sob $=^k$ é o conjunto dos precursores da $(j-1)$-ésima classe de S sob $=_k$ que tem precursores.*

Prova: A primeira afirmação segue de que para cada $u \in S$ temos $u(k+$

$1, 2k) = \lambda$ se e somente se $|u| \leq k$. Quanto à segunda, sejam $C_1, C_2, \ldots, C_n$ as classes de S sob $=_k$ que tem precursores, dispostas segundo a ordem $\preceq_k$.

Seja D_i o conjunto dos precursores de C_i, $i = 1, 2, \ldots, n$. A família $\mathcal{D} = \{D_i; 1 \leq i \leq n\}$ é evidentemente uma partição de $S_k = \{u \in S; |u| > k\}$. Note agora que se $u, v \in S$ são os precursores de $u', v' \in S$, então $u =^k v$ se e somente se $u =_k v$, e portanto $\mathcal{D}$ é a família das classes de S sob $=^k$. Mais ainda, $u \preceq^k v$ se e somente se $u' \preceq_k v'$, donde $D_i \preceq^k D_j$ se $1 \leq i < j \leq n$ e estamos feitos.

A *operação derivada* de que falamos é visitar as classes de S sob $=^k$ (na ordem $\preceq^k$) e, para cada classe visitada, visitar todos os seus elementos. Vejamos como fazê-lo. Começa-se visitando os elementos de $\{u \in S; |u| \leq k\}$. Em seguida visita-se, na ordem $\preceq_k$, as classes de S sob $=_k$ e, para cada uma delas, visitamos o precursor de cada um dos seus elementos que tiver precursor. Essa operação derivada será chamada de o *percurso derivado*, e é nela que se baseia o passo refinador.

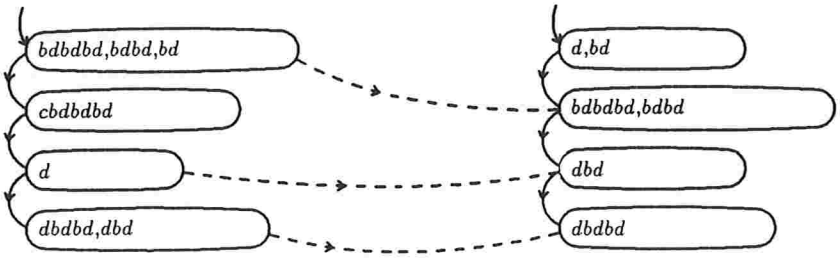


Figura 2.4: O percurso derivado: à esquerda temos as classes de $S = S(cbdbdbd)$ sob $=_2$ na ordem $\preceq_2$. À direita temos as classes de S sob $=^2$. Uma seta tracejada liga cada classe à esquerda com o conjunto dos seus precursores.

O passo refinador consiste, é claro, em particionar as classes de S sob $=_k$ nas suas subclasses sob $=_{2k}$. O algoritmo deverá substituir ao nível da estrutura de dados cada classe C de S sob $=_k$ pela lista das suas subclasses ordenadas segundo $\preceq_{2k}$. Ao fazê-lo obterá automaticamente a lista das classes de S sob $=_{2k}$ ordenadas segundo $\preceq_{2k}$.

Proposição 50 *Seja C uma classe de S sob $=_k$. A i-ésima classe de C sob $=_{2k}$ é a interseção de C com a i-ésima classe de S sob $=^k$ que tem interseção não vazia com C.*

Prova: Sejam $C_1, C_2, \ldots, C_n$ as classes de S sob $=^k$ que tem interseção não vazia com C, dispostas segundo a ordem $\preceq^k$. Seja $D_i = C \cap C_i$ $(1 \leq i \leq n)$. A família $\mathcal{D} = \{D_i; 1 \leq i \leq n\}$ é evidentemente uma partição de C. Dados $u, v \in C$, temos $u =_{2k} v$ se e somente se $u =^k v$, e portanto $\mathcal{D}$ é a família das classes de C sob $=_{2k}$. Por outro lado, $u \preceq_{2k} v$ se e somente se $u \preceq^k v$. Portanto $D_i \preceq D_j$ se $1 \leq 1 < j \leq n$ e estamos feitos.

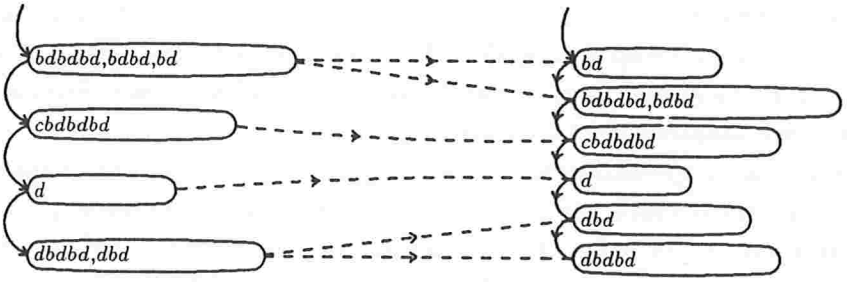


Figura 2.5: O passo refinador: à esquerda temos as classes de $S(cbdbdbd)$ sob $=_2$ na ordem $\preceq_2$. À direita temos as classes de S sob $=_4$. Uma seta tracejada liga cada classe à esquerda com cada uma das suas subclasses.

A proposição 50 garante a correção do passo refinador, que, como já dissemos, usará o percurso derivado. Através dele será visitado cada elemento u de S numa ordem conveniente para construirmos simultâneamente as subclasses sob $=_{2k}$ de cada classe de S sob $=_k$.

para cada classe C **de** S **sob** $=^k$ **faça**
  para cada $u \in C$ **marque** $\overline{u}$
  para cada $u \in C$ **faça**
    se $\overline{u}$ **estiver marcada então**
      desmarque $\overline{u}$
      crie uma nova subclasse de $\overline{u}$
    adicione u **à subclasse atual de** $\overline{u}$

Algoritmo 2.1: O passo refinador. Cada subclasse de $\overline{u}$ criada é colocada no final da lista das subclasses de $\overline{u}$, passando a ser a *subclasse atual de* $\overline{u}$. .

2.2.2 Estruturas de dados

Começaremos agora a descrever uma implementação das listas das classes e do passo refinador que utiliza ao todo dois vetores de $|x|$ inteiros e dois vetores de $|x|$ bits. Para os vetores de bits podemos fazer uso dos sinais das entradas dos dois vetores de inteiros, e por isso consideraremos que o consumo de memória que eles representam é nulo. Nossa implementação do passo refinador consumirá tempo $O(|x|)$, e, como no pior caso necessitamos executar $\lceil \log_2 |x| \rceil$ passos, o tempo total de refinamento será $O(|x| \log |x|)$. Como veremos, o tempo e a memória necessários para obter a base dos refinamentos pode também ser mantida dentro desses limites.

Para simplificar a apresentação, suporemos que os sufixos serão representados *explicitamente* (é claro, entretanto, que na prática o sufixo $x_i x_{i+1} \ldots x_{|x|}$ é representado por i). As listas dos elementos das classes de S sob $=_k$ serão representadas num único vetor f indexado por $\{1, 2, \ldots, |x|\}$ e cujas entradas são elementos de S. A lista das classes estará implícita, visto que as primeiras n_1 entradas de f serão ocupadas pelos n_1 elementos da primeira classe de S sob $=_k$, as n_2 posições seguintes pelos n_2 elementos da segunda classe, e assim por diante. O vetor de bits p (indexado por $\{1, 2, \ldots, |x|\}$) será utilizado para marcar os inícios das listas (veja a figura 2.6).

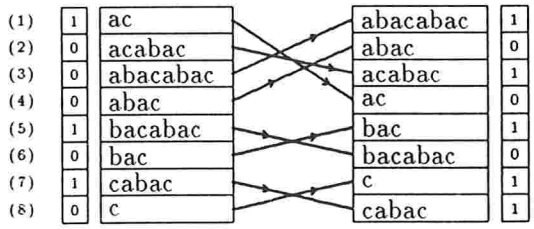


Figura 2.6: O par f, p antes (esquerda) e depois (direita) do passo refinador que, a partir das classes de $S = S(abacabac)$ sob $=_1$ obtém as classes de S sob $=_2$.

Os índices das entradas de p que valem 1, juntamente com $|x| + 1$, serão chamados de as *k-lideranças* ou apenas *lideranças* (de x). Se j e j' forem respectivamente a i-ésima e a $(i+1)$-ésima lideranças, então por definição $C = \{f[j], f[j+1], \ldots, f[j'-1]\}$ é a i-ésima classe de S sob $=_k$, e para todo

$u \in C$ diremos que j é *a liderança* de u. Diremos também que j é a *liderança* de C.

Postas essas estruturas de dados, o passo refinador pode ser visto como uma permutação das entradas de f. De fato, o efeito da chamada de **Refine** (algoritmo 2.3) será uma permutação das entradas de f e a alteração de 0 para 1 de algumas entradas de p (aquelas indexadas por $2k$-lideranças que não são k-lideranças). Chamaremos o índice da entrada de f que armazena $u \in S$ após a chamada de **Refine** de *nova posição de u*.

No passo refinador é necessário, dado um elemento $u \in S$, determinar a classe $\overline{u}$. Para que isso possa ser feito em tempo constante, calcularemos previamente essa informação, fazendo uso de um vetor de inteiros g indexado por S. A entrada de g indexada por u será inicializada com a liderança de u pelo algoritmo 2.2.

A implementação das listas através de um vetor faz necessários contadores para que a inserção no final de cada lista seja feita em tempo constante. A fim de implementar o passo refinador, será suficiente alocar, para cada classe de S sob $=_k$, exatamente um contador. Vejamos agora como fazer isso.

Poderíamos alocar um vetor de inteiros indexado por $\{1, 2, \ldots, |x|\}$ e usar a entrada de índice $g[u]$ como sendo o contador da classe $\overline{u}$. Entretanto já alocamos dois vetores de tamanho $|x|$ (f e g), ambos (na prática) de inteiros, e a alocação de mais um elevaria o consumo de memória para $3|x|$ inteiros (é isso que é feito em [19]).

Para mantermos a memória total em $2|x|$ inteiros, será necessário utilizar, para implementar os contadores, parte do vetor g. Para cada classe C de S sob $=_k$ escolheremos um elemento $u \in C$ que será chamado de o *líder de* C, e usaremos $g[u]$ para armazenar o contador da classe $\overline{u}$. Note que necessitaremos, dado um índice u, distinguir se $g[u]$ é um contador ou se armazena o liderança da classe $\overline{u}$. O vetor de bits q indexado por S irá encarregar-se disso: no primeiro caso teremos $q[u] = 1$ e, no segundo, $q[u] = 0$ (veja a figura 2.7).

Vejamos como fazer a escolha dos líderes. A fim de que o contador de $\overline{u}$ possa ser acessado em tempo constante a partir de u, faremos com que o líder de $\overline{u}$ seja precisamente o sufixo que ocupa a liderança de $\overline{u}$ (daí o nome *líder*). Portanto o contador da classe $\overline{u}$ será $g[f[g[u]]]$ se u não for líder, ou $g[u]$ caso contrário (no caso de u ser líder estaremos impossibilitados de obter em tempo constante a liderança de u).

2.2.3 Estabilização

Resta-nos ainda uma dificuldade para resolver. O vetor f não pode ser alterado durante a execução do passo refinador, pois necessitamos dele intacto para levar o percurso derivado até o fim e também para sermos capazes de acessar o contador de $\overline{u}$ quando u não for líder. Assim, a permutação de f que o passo refinador obtém terá que ser armazenada noutro lugar. Como não podemos alocar mais memória, usaremos novamente o vetor g.

Para cada $u \in S$ que não é líder, necessitamos de $g[u]$ apenas no momento em que u é visitado no percurso derivado (para acessarmos o contador $g[f[g[u]]]$). A partir desse momento, portanto, podemos dispor de $g[u]$, que será usada para armazenar a nova posição de u.

```
1    Procedimento Estabilize
2      j ← 1; q[1 ... |x|] ← 0
3      para i de 1 até |x| faça
4        se p[i] = 1 então j ← i
5        gf[i] ← j
6      para todo i de −p + 1 até |x| com t_i ≠ λ faça
7        se pg[t_i] = 1 então
8          pg[t_i] ← 0; q[t_i] ← 1
9      para todo i de 1 até |x| com q[i] = 1 faça
10         pg[i] ← 1; j ← g[i]
11         enquanto f[j] ≠ i faça j ← j + 1
12         fg[i] ↔ f[j]; g[i] ← 0
```

Algoritmo 2.2: O procedimento de estabilização.

Se u for líder, não poderemos dispor dessa forma de $g[u]$, pois esse é o contador de $\overline{u}$. Para isso teremos que lançar mão de uma *hipótese de estabilidade*. Suporemos que, para cada classe C de S sob $=_k$, o líder u de C satisfaz $u \preceq_{2k} v$, todo $v \in C$. Isso significa que podemos assumir, de antemão, que a nova posição de u é a liderança de u. Assim essa nova posição não precisará ser armazenada.

A hipótese de estabilidade pode ser garantida facilmente com um preprocessamento de f. Aproveitaremos esse preprocessamento para inicializar

os vetores g e q, e também para exemplificar a implementação do percurso derivado a partir das estruturas de dados que adotamos.

A inicialização de $g[u]$ com a liderança de u, todo $u \in S$ é trivialmente feita no laço 3-5. Na linha 6 temos o controle do percurso derivado. O símbolo t_i representa $x_{|x|-i}x_{|x|-i+1}\ldots x_{|x|}$ se $i \leq 0$ ou o precursor de $f[i]$ caso contrário (se $f[i]$ não tiver precursor, convencionamos que t_i vale λ).

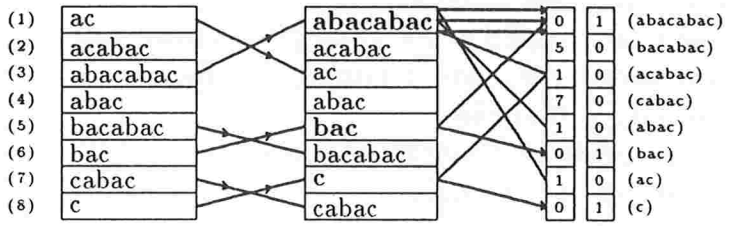


Figura 2.7: À esquerda temos o vetor f representando as classes de $S = S(abacabac)$ sob $=_1$ antes e depois da estabilização. Os líderes estão em destaque. À direita temos os vetores g e q com a indicação, para cada $u \in S$ não líder, de $g[f[g[u]]]$.

Para cada classe C de S sob $=_k$, **Estabilize** obtém o primeiro elemento de C visitado pelo percurso derivado. Assim que isso ocorrer, zeramos $pg[u]$ (linha 8) para sabermos que um elemento de C já foi visitado e marcamos $q[u]$ (linha 8 também) para sabermos que u é líder.

No laço 11-12, f é alterado para que os líderes escolhidos ocupem efetivamente as entradas de f indexadas pelas lideranças. Além disso restauramos o vetor p e zeramos $g[u]$ para todo líder u, pois $g[u]$ será o contador de $\overline{u}$. É fácil ver que o **enquanto** da linha 11 nunca é executado duas vezes com o mesmo valor de j, e portanto o tempo total de **Estabilize** é $O(|x|)$.

2.2.4 Implementação do passo refinador

Podemos, finalmente, dar a implementação do passo refinador. Ele começa (linha 2) executando o preprocessamento que garante a hipótese de estabilização e inicializa g e q. Em seguida faz-se o que já havia sido descrito no algoritmo 2.1. Ao término da execução da linha 17, g^{-1} será o vetor que

armazena as listas das classes de S sob $=_{2k}$, e teremos $q[u] = 1$ se e somente se $g[u]$ for uma $2k$-liderança.

```
1    Procedimento Refine
2      Estabilize
3      i ← −p + 1
4      enquanto i ≤ |x| faça
5        f ← min({i ≤ k < |x|; p_{k+1} = 1} ∪ {|x|})
6        para j de i até f faça
7          se t_j ≠ λ e q[t_j] = 0 e gfg[t_j] > 0 então pg[t_j] ← 0
8        para todo j de i até f com t_j ≠ λ faça
9          se q[t_j] = 0 então
10           se gfg[t_j] = 0 então gfg[t_j] ← 1
11           se pg[t_j] = 0 então
12             pg[t_j] ← 1;q[t_j] ← 1
13           g[t_j] ← g[t_j] + gfg[t_j]
14           gfg[t_j] ← gfg[t_j] + 1
15         senão se g[t_j] = 0 então g[t_j] ← 1
16       i ← f + 1
17     para todo i de 1 até |x| com p[i] = 1 faça gf[i] ← i
18     para i de 1 até |x| faça fg[i] ← i
19     para i de 1 até |x| faça p[i] ← qf[i]
```

Algoritmo 2.3: Implementação do passo refinador.

Assim que a linha 5 é executada, a classe de S sob $=^k$ em visita está definida pelo par (i, f): ela é $\{t_j; i \leq j \leq f\}$, desde que esse conjunto seja não vazio. Na linha 7 marcam-se as classes de S sob $=_k$ que têm interseção não vazia com $\{t_j; i \leq j \leq f\}$. Para isso usa-se, como fizemos em **Estabilize**, o vetor p, que logo em seguida (linha 12) será restaurado.

A finalidade da marcação das classes é, como vimos no algoritmo 2.1, a alocação de uma nova subclasse. Na nossa estrutura de dados, isso significa que um determinado índice deve ser distinguido como $2k$-liderança, o que é feito na linha 12 pela atribuição $q[t_j] \leftarrow 1$. Evita-se fazer essa atribuição quando essa $2k$-liderança já for uma k-liderança, e é por isso que na linha 7

testa-se $g[t_j]$ contra zero. De fato, uma análise mais cuidadosa do algoritmo mostra que o primeiro elemento de cada classe de S sob $=_k$ visitado pelo percurso derivado pode não ser o líder da classe, e por isso na linha 12 alteraríamos q erradamente se não tratássemos separadamente as k-lideranças.

Nas linhas 13 e 14 faz-se a inserção de t_j na subclasse atual, desde que t_j não seja líder. Isso corresponde ao cálculo da nova posição de t_j a partir da sua liderança e do contador $g[f[g[t_j]]]$, que em seguida é incrementado. Nas linhas 10 e 15 faz-se a inserção do líder na subclasse atual. Como não podemos armazenar em g a nova posição dos líderes, apenas incrementa-se o contador da classe. Na linha 17 as novas posições dos líderes (ou seja, as k-lideranças), são obtidas explicitamente. Em seguida procede-se à atribuição $f \leftarrow g^{-1}$ (linha 18) e à marcação das $2k$-lideranças (linha 19). É imediato que o tempo total de **Refine** é $O(|x|)$.

2.2.5 A Base

Resta-nos fazer alguns comentários acerca da obtenção da base dos passos refinadores, isto é, as classes de S sob $=_1$. Na nossa estrutura de dados, isso significa inicializar f com uma permutação de S de tal forma que valha $f[i] \preceq_1 f[j]$ sempre que $1 \leq i < j \leq |x|$. Ora, isso equivale à ordenação de um vetor de $|x|$ letras. De fato, é suficiente inicializarmos $f[i]$ com $x_i x_{i+1} \ldots x_{|x|}$ e ordenarmos f segundo as primeiras letras das suas entradas.

Como no refinamento usamos apenas $2|x|$ inteiros e tempo total $O(|x| \log |x|)$, seria interessante que essa ordenação se mantivesse dentro desses limites. Sabemos que f será na verdade um vetor de inteiros, e portanto ele representa um consumo de $|x|$ inteiros. Os outros $|x|$ inteiros poderão ser utilizados para armazenar x durante a obtenção da base [3] (é desnecessário dispor de x durante o refinamento). Assim, precisamos de um algoritmo de ordenação $O(|x| \log |x|)$ que não use espaço adicional. Em [19] sugere-se o radix-sort, mas, ao contrário do que se diz lá, isso exige a hipótese de que o tamanho do alfabeto seja limitado por uma constante a fim de que o tempo total (que nesse caso seria $O(|x|)$) não dependa do tamanho do alfabeto.

Uma possibilidade é alguma implementação não-recursiva do mergesort que não utilize memória auxiliar para as intercalações (veja por exemplo [14]). Deve-se notar, entretanto, que nesse caso o tempo médio da obtenção

[3] Assim como no caso do autômato dos sufixos, suporemos por simplicidade que não é necessário ler x.

da base será $O(|x| \log |x|)$, enquanto o tempo médio total do refinamento sob a hipótese de distribuição uniforme é $O(|x| \log\log |x|)$, pois nessas condições o comprimento do fator repetido mais longo de x é logarítmico em $|x|$ [16].

2.3 Um algoritmo de ordenação externa

Passemos agora a um algoritmo desenvolvido por Gonnet et alli [12] durante o projeto de informatização do Dicionário de Oxford (OED). Ele é quadrático no quociente do comprimento da entrada pelo tamanho da memória principal disponível; o tamanho atual das memórias permite que ele seja utilizado sem maiores problemas para textos da ordem de grandeza do OED.

2.3.1 Índices parciais

Nesse contexto despreocupar-nos-emos com o tamanho do alfabeto, que assumiremos ser o alfabeto romano, e alteraremos ligeiramente a nossa terminologia, a fim de aproximá-la da usada em linguagens naturais. Ao invés de *palavra*, chamaremos x de *texto*. O termo *palavra* passará a ter o mesmo significado que nas linguagens naturais (as palavras do texto "ser ou não ser" são "ser", "ou" e "não").

Em virtude do tamanho do texto (570 megabytes no caso do OED), não ordenaremos S, mas um subconjunto próprio de S: serão considerados apenas os *sufixos úteis*. Um sufixo $x_i x_{i+1} \ldots x_{|x|}$ é útil quando i é uma *posição útil*. Na informatização do OED, as posições úteis são os *inícios de palavras*, que no texto "ser ou não ser" são $1, 5, 8$ e 12". Outra restrição que faremos é que a ordenação lexicográfica ficará restrita às L primeiras letras. Na prática um valor pequeno para L (50, por exemplo) decide a grande maioria das comparações. No caso do OED as comparações não resolvidas foram tratadas à parte. Na nossa apresentação não tomaremos esse cuidado.

O algoritmo que descreveremos admite qualquer escolha das posições úteis[4], desde que, dado $i \in \{1, 2, \ldots, |x|\}$, x_i, e, eventualmente algumas outras letras na sua vizinhança (como x_{i-1} por exemplo), seja possível decidir se i é útil ou não.

[4] O algoritmo de Manber e Myers só admite uma escolha, a saber, *todo elemento de* $\{1, 2, \ldots, |x|\}$ *é posição útil.*

Seguiremos um esquema de ordenação externa. Dado um subconjunto S' de S, uma função bijetora $f\colon \{1,2,\ldots,|S'|\} \to S'$ que satisfaça $f(i) \preceq_L f(j)$ quando $1 \leq i < j \leq |S'|$ será chamada um *L-índice* de S' (lembramos que $\preceq_L$ denota a ordem lexicográfica restrita aos L primeiros caracteres). O conjunto dos sufixos úteis $U = U(x)$ de x será particionado e construiremos um L-índice de cada uma das classes, para em seguida intercalá-los e assim obter um *L-índice para o texto*, relativo à escolha feita dos sufixos úteis. Na descrição do algoritmo, a implementação dos L-índices será feita diretamente por vetores de inteiros.

```
1   Procedimento Índice
2   k ← 0; n ← 0
3   enquanto kM < |x| faça
4     m ← 0; k ← k + 1
5     para cada i ∈ {(k − 1)M + 1, (k − 1)M + 2, ..., kM} útil faça
6       m ← m + 1; f[m] ← i
7     se m > 0 então
8       Ordene(f)
9       c[0 ... k] ← 0
10      para cada i ∈ {1, 2, ..., (k − 1)M} útil faça
11        l ← pos(f, i); c[l] ← c[l] + 1
12      escreva(índices[k], u[1], u[2], ..., u[m])
13      escreva(contadores[k], c[1], c[2], ..., c[m])
14      precedentes[m] ← c[0]; n ← n + m
15    senão precedentes[m] ← n
16  se n > 0 então Remova(m, n)
```

Algoritmo 2.4: Algoritmo de ordenação externa para a construção do vetor dos sufixos

O *i-ésimo bloco* (de x), onde $1 \leq i \leq k$, será o fator $x((i-1)M+1, iM)$, onde M é uma constante que depende da memória principal disponível. A classe U_i será o conjunto dos sufixos úteis $x_i x_{i+1} \ldots x_{|x|}$ que satisfazem $(i-1)M < i \leq iM$, onde novamente temos $1 \leq i \leq k$. Para as nossas estimativas do desempenho assumiremos que U_i tem no máximo $M/10$ elementos. Como veremos logo mais, essa hipótese permite-nos considerar que M é metade

da memória principal medida em bytes. O total de classes é, naturalmente, $k = \lceil |x|/M \rceil$.

Vejamos algo sobre o desempenho. Nesse ponto é necessário separar as operações internas das operações de entrada e saída. Quanto às primeiras, consomem tempo $O(|x|^2 L \log M)/M$. Por sua vez, as operações de entrada e saída não lêem ou escrevem mais do que $|x|^2/(2M) + 2|x|$ caracteres. Vale observar que o algoritmo é parcialmente paralelizável.

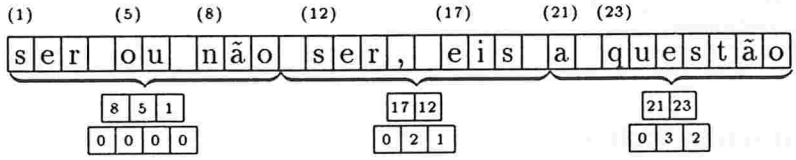


Figura 2.8: Índices e contadores: as posições úteis – e só elas – estão indicadas sobre o texto. As chaves identificam os blocos ($M = 10$). Sob cada bloco damos um 3-índice e um vetor de contadores.

Admitindo que o i-ésimo bloco (juntamente com os $L-1$ caracteres de x que lhe seguem) está na memória principal, podemos empregar qualquer algoritmo de ordenaccão para obter um L-índice f_i de U_i (no caso do OED usou-se o QuickSort). Assumiremos que o tempo dispendido nessa operação é $O(LM \log M)$. Supondo que usemos inteiros de quatro bytes, temos um consumo máximo de 1.8 bytes por letra do bloco para armazenar o bloco, um L-índice seu e um vetor de $|U_i|+1$ contadores que será necessário para a intercalação. Assim, podemos tomar para M metade da memória principal medida em bytes, como dissemos anteriormente.

2.3.2 Intercalação

A intercalação dos n L-índices não é trivial, e para conseguir realizá-la com alguma eficiência, será necessário, logo após construirmos o L-índice de U_i, reler $x(1,(i-1)M)$ e construir um vetor de contadores da forma que segue. Com os contadores todos inicialmente zerados, para cada sufixo útil $u = x_j x_{j+1} \ldots x_{|x|}$ com $j \leq (i-1)M$ teremos que determinar $l \in \{0,1,\ldots,|U_i|\}$ satisfazendo $f_i(l) \preceq_L u \preceq_L f_i(l+1)$ ($f_i(0)$ e $f_i(|U_i|+1)$ são sentinelas). Isso pode ser facilmente conseguido com uma busca binária. Obtido l, incrementa-se o l-ésimo contador que, ao final, armazenará o número

de sufixos úteis das classes $U_1, U_2, \ldots, U_{i-1}$ que precederão $f_i(l)$ (mas não $f_i(l+1)$) na intercalação.

Note que apesar de relermos $(i-1)M$ caracteres, a cada momento precisamos ter apenas L caracteres na memória além do i-ésimo bloco. Obtidos f e o vetor de contadores, armazenamo-os nos arquivos de inteiros *índices*[i] e *contadores*[i]. O tempo total para a construção dos L-índices é $O(L|x|\log M)$. A construção do i-ésimo vetor de contadores necessita tempo $O(iML\log M)$, que, somado para todo i, leva-nos a $O((|x|^2 L\log M)/M)$. Note que a construção do i-ésimo L-índice e do respectivo vetor de contadores exige a leitura de iM caracteres e a escrita de no máximo $M/5$ inteiros (ou, equivalentemente, $4M/5$ caracteres). Somando-os para todo i, temos um máximo de $4|x|/5 + |x|^2/(2M)$ caracteres lidos ou escritos.

```
1   Procedimento Remova(i, t)
2   repita
3     se precedentes[i] > 0 então
4       t' ← min{t, precedentes[i]}
5       remova(i − 1, t')
6       t ← t − t'
7       precedentes[i] ← precedentes[i] − t'
8     senão
9       leia(índice[i], j)
10      escreva(Índice,j)
11      leia(contadores[i],precedentes[i])
12      t ← t − 1
13  até que t = 0
```

Algoritmo 2.5: A rotina de intercalação.

Inicialmente (linhas 5-6) **Índice** obtém U_k e, após um teste de consistência, construímos um L-índice para ele (linha 8). Inicializamos o vetor de contadores c (linha 9) e, para cada posição útil em $\{1, 2, \ldots, (k-1)M\}$ obtemos l como descrito anteriormente (**pos** realiza a busca binária) e incrementamos $c[l]$. Os vetores f e c são então armazenados em arquivos. A primeira entrada de c é utilizada para inicializar *precedentes*[k]. O papel do

vetor *precedentes* ficará claro na rotina de intercalação **Remova**, chamada na linha 16 desde que o total n de sufixos úteis não seja zero. A memória adicional consumida por esse vetor pode ser desconsiderada, pois a informação que ele contém só necessita estar na memória principal durante o intercalamento.

O efeito da chamada *Remova*(i,t) é ler dos índices $1,2,\ldots,i-1$ t elementos. Não se pode dizer que esses elementos lidos sejam os *menores* que se encontram nos índices, mas os elementos que restarem naqueles índices não precedem estritamente qualquer um desses t lidos relativamente a $\preceq_L$. Da construção dos vetores de contadores segue rapidamente que a chamada de *Remova* realizada em **Índice** procede corretamente à intercalação dos índices parciais. O resultado da intercalação é armazenado no arquivo de inteiros *Índice*.

Classificaremos as chamadas de **Remova** em dois tipos. O primeiro será composto pelas chamadas em que a linha 9 é executada. Como há no máximo $|x|/10$ sufixos úteis, há no máximo $|x|/10$ dessas chamadas. O segundo será formado pelas outras chamadas. Vejamos quantas destas pode haver. Como $t' > 0$ na linha 6, o total dessas chamadas não pode superar a soma das entradas de todos os vetores de contadores. Ora, essa soma não supera $|x|^2/(20M)$, pois cada sufixo útil de U_i é contabilizado exatamente $k-i$ vezes ao longo das construções dos contadores. Assim o tempo total consumido pela intercalação é $O(|x|^2/M)$. Quanto às operações de entrada e saída, no máximo $3|x|/10$ inteiros são lidos ou escritos.

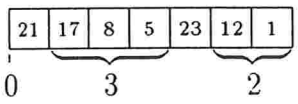


Figura 2.9: Resultado da intercalação dos índices parciais da figura 2.8. A seqüência dos comprimentos dos intervalos que separam os elementos do terceiro índice entre si ou das extremidades formam o terceiro vetor de contadores $(0,3,2)$ daquela figura.

Portanto o tempo das operações internas é assintóticamente majorado pelo tempo de construção dos vetores de contadores, que é $O((|x|^2 L \log M)/M)$. Por outro lado, a soma de todos os caracteres lidos ou escritos por **Índice** (no máximo $4|x|/5 + |x|^2/(2M)$) e por **Remova** (no máximo $6|x|/5$) não supera $|x|^2/(2M) + 2|x|$.

Capítulo 3

Ferramentas Computacionais em Biologia Molecular

Recentes avanços em biologia molecular vêm revelando a estrutura primária de diversas proteínas, e a seqüência de bases de trechos do DNA de inúmeros organismos. O volume de informações vem crescendo com grande velocidade, e o uso de computadores nessa área é cada vez mais intenso. Com eles têm-se organizado grandes bancos de dados e desenvolvido diversas ferramentas para a análise dessas *seqüências biológicas*. Os problemas que surgem então não são triviais, e têm pelo menos dois agravantes, que são as enormes dimensões dos bancos de dados biológicos e a ocorrência de erros de seqüenciamento.

Esses problemas têm chamado a atenção dos cientistas de computação. Algo que pode ajudar-nos a medir o nível de interação atual entre a biologia molecular e a ciência da computação é o número de entradas de algumas bibliografias recentemente organizadas sobre o tema; nas maiores de que temos notícia esse número gira em torno de mil [13].

O esforço de pesquisa tem trazido benefícios para os dois lados. Naturalmente, boa parte dos resultados obtidos têm um interesse ou uma aplicação muito particulares. Os métodos para a detecção de homologias, por exemplo – e é necessário que o façam –, são calibrados em função de determinadas características estruturais das seqüências biológicas. Essas características funcionam como hipóteses simplificadoras que, por um lado, permitem ganhos de eficiência, mas por outro tornam o método inaplicável noutros contextos.

Pode-se entretanto citar alguns trabalhos que extrapolam essa especificidade, constituindo-se por isso em legítimas contribuições para a ciência da

computação. O vetor dos sufixos, por exemplo, é, como já dissemos, em parte fruto dessa interação. Outros exemplos são um algoritmo de aproximação para o problema da reconstrução [3] e diversas estatísticas [17] envolvendo seqüências.

De maneira geral, entretanto, convém não perder de vista que a interdisciplinaridade aqui é inevitável. Longe de ser um transtorno, essa exigência é com certeza bastante salutar. É essa a tônica que tentamos dar a este capítulo. Expusemos várias ferramentas computacionais usadas em biologia molecular procurando situar cada uma delas dentro do contexto particular em que foi desenvolvida. Nossa fonte principal para os conceitos básicos de bioquímica foi o livro de Lubert Stryer [26]. Sempre que possível relacionamos essas ferramentas com as estruturas baseadas em sufixos já apresentadas. Nossa abordagem não é exaustiva; entre os tópicos que não cobrimos pode-se citar o problema da reconstrução, a previsão da estrutura espacial de proteínas, o cálculo de alinhamentos e a detecção de consensos.

3.1 Detecção de Homologias

Uma *homologia* entre dois genes é, no sentido próprio do termo, a existência de um ancestral comum de onde os dois originaram-se. As mutações ocorridas ao longo dos séculos fazem com que genes homólogos tornem-se cada vez mais distintos. A semelhança residual, entretanto, é suficiente para detetarmos a homologia por métodos computacionais.

A formalização da noção de semelhança entre duas seqüências pressupõe um *modelo* composto de operações sobre seqüências com custos associados. O estudo dos diferentes modelos subjacentes aos inúmeros métodos de detecção de homologias já propostos fornece um ponto de vista unificado para as suas avaliações.

Um modelo fundamental utilizado nesse contexto é aquele implícito no chamado *problema do cálculo da subseqüência comum mais longa*, de larga aplicação em diversas áreas [24]. A partir dele desenvolveu-se inúmeras implementações, entre elas o programa FASTA. Há aí entretanto uma dificuldade difícil de ser superada, que é a aparente inexistência de algoritmos suficientemente rápidos (os melhores conhecidos são essencialmente quadráticos). Uma simplificação desse modelo, que abrange apenas um tipo de mutação – o mais freqüente de todos –, foi implementado no programa BLAST.

3.1.1 Homologias

As *proteínas* são longas cadeias de *aminoácidos*. Cada um dos vinte aminoácidos que ocorrem nas proteínas é representado por uma letra do alfabeto romano (*alanina* por *a*, *arginina* por *r*, etc). A esse alfabeto de vinte letras costuma-se adicionar outras três que têm significados especiais.

As cadeias de aminoácidos têm uma *orientação* [1] bem definida, e por isso cada proteína pode ser representada pela seqüência das letras que representam os seus aminoácidos. Em outras palavras, AAL e LAA representam cadeias distintas.

Desde o seqüenciamento da insulina por Frederick Sanger em 1953, já se obteve a cadeia de aminoácidos de milhares de proteínas. A descoberta, por vezes casual, de *semelhanças* entre duas ou mais dessas cadeias, motivou o desenvolvimento de métodos computacionais para a detecção de homologias. Dadas duas seqüências, esses métodos tentam determinar se é ou não possível transformar uma na outra com um número *relativamente pequeno* de operações elementares como trocas, inserções e deleções de caracteres.

```
130        140        150        160        170            180
DLRAANILVGENLVCKVADFG----LARLIEDNEYTARQGAKFPIKWTAPEAALYGRFTI
X:..:::::.:. ::::.::X    :. : . .. ... :.... .  :::    .  :
DLKPANILISEQDVCKISDFGCSQKLQDLRGRQASPPHIGGTYTHQ--APEILKGEIATP
   150        160        170        180        190          200
```

Figura 3.1: Um trecho de uma homologia entre duas cadeias de aminoácidos detectada pelo FASTA. Os tracejados indicam *lacunas*, e são inseridos para que os aminoácidos correspondentes disponham-se nas mesmas colunas. A figura que se obtém dessa forma chama-se um *alinhamento*.

As proteínas são sintetizadas pelas células indiretamente a partir das informações contidas no *ácido desoxirribonucleico (DNA)*. As moléculas de

[1] Para formar cadeias, cada aminoácido conta com um grupo amino ($^{+}NH_3$) e com um grupo carboxil (COO^{-}). Nas ligações entre aminoácidos contíguos de uma cadeia, um comparece com o grupo amino e o outro com o grupo carboxil. Assim, um dos dois aminoácidos das extremidades está com o seu grupo amino livre de ligações. Este é, por convenção, o início da cadeia.

DNA são longas cadeias de *nucleotídeos*. Na composição de cada nucleotídeo entra uma de quatro *bases*, a saber, *adenina* (a), *citosina* (c), *guanina* (g) e *timina* (t). Assim como as proteínas, as moléculas de DNA também possuem uma orientação [2] , e por isso podem ser representadas por palavras sobre $\{a, c, g, t\}$. Daqui por diante A denotará um dos dois alfabetos de que falamos, o dos aminoácidos ou o das bases.

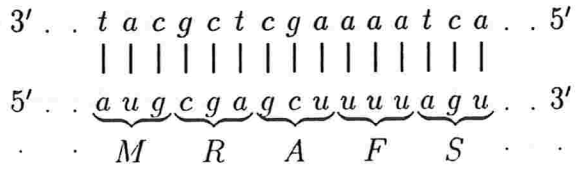


Figura 3.2: Um fragmento de DNA e a seqüência de aminoácidos correspondente. Entre as duas vê-se o mRNA que intermedeia a síntese proteica. Estruturalmente o mRNA é semelhante ao DNA, e as bases que comparecem na sua composição são as mesmas que no DNA, exceto pela timina, que é trocada pela uracila (U).

Três letras contíguas no DNA formam um *códon*. Cada códon determina um aminoácido no processo de síntese proteica. A regra que associa cada códon ao aminoácido por ele determinado chama-se *código genético*. Um trecho de DNA que contém as informações necessárias para a síntese de uma proteína (ou de moléculas de RNA que não chegam a expressar proteínas, como no caso do tRMA) é chamado *gene*. Dessa forma, ao invés de procurarmos homologias nas proteínas, podemos, ao menos em princípio, procurá-las diretamente nos genes.

Há uma série de detalhes, entretanto, que fazem com que a busca de homologias nas proteínas e nos genes sejam processos distintos. O primeiro é que o código genético é *degenerado*, isto é, há aminoácidos que podem ser determinados por mais de um códon, e portanto uma homologia ao nível das

[2] Para formar cadeias, cada nucleotídeo conta com duas hidroxilas (OH^-), chamadas 3′ e 5′ por estarem ligadas aos carbonos 3′ e 5′ da desoxirribose que, unida à base, forma o nucleotídeo. Nas ligações entre aminoácidos contíguos de uma cadeia, um comparece com a 3′-hidroxila e o outro com a 5′-hidroxila. Assim, um dos dois nucleotídeos das extremidades está com a sua 5′-hidroxila livre de ligações. Este é, por convenção, o *início* da cadeia.

proteínas não acarreta necessariamente uma homologia ao nível dos genes. A recíproca também pode ocorrer, já que, por exemplo, duas seqüências de bases que diferem apenas pela inserção de uma base no começo de uma delas podem codificar proteínas completamente disferentes, pois com isso altera-se as posições de início dos códons. Outro motivo é o fato de que o mRNA em geral sofre alterações antes de ser efetivamente utilizado na síntese proteica. Uma última observação é que a busca de homologias em proteínas conta com um modelo estocástico desenvolvido por Dayhoff para estimar o quão significativa é uma *semelhança* formalmente detectada entre duas cadeias (trata-se das matrizes PAM).

3.1.2 Índices de Similaridade

Um *índice de similaridade* é, intuitivamente, uma função que quantifica a *semelhança* entre duas seqüências [3]. Quando o índice de similaridade entre duas seqüências for alto (segundo algum critério estatístico), diremos que essas duas seqüências são *similares* relativamente a esse índice. No caso de seqüências biológicas, uma similaridade no sentido formal do termo pode ou não ter importância biológica. Em particular, ela pode ou não dever-se a uma homologia.

Um índice de similaridade deve creditar aquilo que duas seqüências têm em comum e debitar aquilo em que elas diferem. Essas duas noções, isto é, a parte comum a duas seqüências e a diferença entre elas serão derivadas, nos índices que apresentaremos, das *operações* admitidas para transformação de seqüências, e dos custos associados a cada uma delas. No caso da biologia molecular, as operações admitidas são aquelas que modelam as possíveis mutações gênicas, e os seus custos são deduzidos das freqüências com que essas mutações são observadas ou preditas.

Vamos ao nosso primeiro índice, em que admitiremos unicamente a operação *troca de caracteres*. Dados $x, y \in A^*$ com $|x| = |y|$, para cada $i \in \{1, 2, \ldots, |x|\}$, se $x_i = y_i$ creditaremos essa coincidência, e se $x_i \neq y_i$ então debitaremos o custo da troca de x_i por y_i. Denotando por $s(a, a)$ o crédito da coincidência de a e por $-s(a, b)$ o custo da troca [4] de a por b, podemos

[3] Pode-se pensar que duas seqüências são tão mais semelhantes quanto menor for a distância entre elas segundo uma métrica fixada, e, de fato, muitos índices de similaridade podem ser dualizados para métricas.

[4] Um *custo* para nós é um número positivo, daí a inversão do sinal, que simplificará a

definir o *índice de similaridade baseado em trocas* $s_t(x,y) = \sum_{i=1}^{|x|} s(x_i, y_i)$.

O fato do índice s_t estar indefinido quando as seqüências têm comprimentos diferentes não chega a ser preocupante, uma vez que a mutação que ocorre com maior freqüência é a troca. Inserções e deleções entretanto podem ser adicionadas ao modelo sem maiores dificuldades. Dados $x, y \in A^*$, inicialmente escolha um subconjunto $D \subseteq \{1, 2, \ldots |x|\}$ e *delete* as letras $x_i, i \in D$ obtendo z com $|z| = |x| - |D| \leq |y|$. Em seguida execute em z exatamente $|y| - |z|$ *inserções* do caracter $\perp \notin A$ em posições quaisquer, obtendo w. A *similaridade relativa* às deleções e inserções que fizemos é então definida como $s_t(w,y) + \sum_{i \in D} s(x_i, \perp)$ onde, para cada letra a, $-s(a,\perp) > 0$ e $-s(\perp, a) > 0$ são, respectivamente, os custos de deleção e de inserção da letra a.

É fácil ver que, de forma análoga ao que ocorria no índice s_t, estamos creditando as coincidências e debitando as trocas deleções e inserções. Consideradas todas as possíveis escolhas para as deleções e inserções, o valor máximo da similaridade relativa, será, por definição, o *índice de similaridade baseado em trocas, deleções e inserções* $s_d(x,y)$.

		a	t	g	c	t	c	t	t	g	c	a
	0	0	0	0	0	0	0	0	0	0	0	0
c	0	0	0	0	1	1	1	1	1	1	1	1
a	0	1	1	1	1	1	1	1	1	1	1	2
a	0	1	1	1	1	1	1	1	1	1	1	2
t	0	1	2	2	2	2	2	2	2	2	2	2
c	0	1	2	2	3	3	3	3	3	3	3	3
t	0	1	2	2	3	4	4	4	4	4	4	4
t	0	1	2	2	3	4	4	5	5	5	5	5

Figura 3.3: Exemplo de aplicação algoritmo de Needleman-Wunsch. Adotamos $s(a,b) = 1$ quando $a = b$, e 0 quando $a \neq b$. Nessas condições, o cálculo de $s_d(x,y)$ equivale ao cálculo do comprimento de uma *subseqüência comum mais longa* entre x e y.

O índice s_d pode ser calculado por um esquema simples de programação

definição do índice s_t. Em biologia molecular pode ser conveniente considerar que o custo de determinadas trocas que ocorrem com alta freqüência seja negativo; essas trocas são chamadas *trocas conservativas*.

dinâmica em tempo $O(|x||y|)$ e espaço $O(\min(|x|,|y|))$. Em biologia molecular, esse esquema é freqüentemente chamado de *Algoritmo de Needleman-Wunsch*: convencione $M[i,j] = -\infty$ se $i < 0$ ou $j < 0$, ponha $M[0,0] = 0$ e $M[i,j] = \max\{M[i-1,j] + s(x_i, \perp), M[i,j-1] + s(\perp, y_j), M[i-1,j-1] + s(x_i, y_j)\}$, onde $(i,j) \neq (0,0), 1 \leq i \leq |x|, 1 \leq j \leq |y|$. A prova de que $M[i,j] = s_d(x_1x_2 \ldots x_i, y_1y_2 \ldots y_j)$ é trivial.

Na prática, as escolhas para s são funções simétricas [5], e põe-se $s(\perp, a) = s(a, \perp) = -d$ para todo a, onde d é o *custo por deleção*. Essas hipóteses fazem com que os índices s_t e s_d sejam simétricos, e por isso não será necessário especificar se a similaridade refere-se à transformação de x em y ou à transformação de y em x. Além da troca, da deleção e da inserção, alguns trabalhos admitem ainda outras operações [9].

A partir de s_t e de s_d, definiremos agora dois *índices de similaridade locais*. Em biologia molecular, as similaridades ocorrem freqüentemente de forma *localizada*, isto é, encontramos fatores de uma seqüência similares a fatores de uma outra, ainda que as duas seqüências não possam ser consideradas, no seu todo, similares.

Se f for um índice de similaridade definido no subconjunto D_f de $A^* \times A^*$, então o *índice de similaridade local* obtido a partir de f é a função $f_l : (x,y) \mapsto max\{f(u,v); (u,v) \in (\mathrm{Fat}(x) \times \mathrm{Fat(y)} \cap D_f\}$. Os índices de similaridade locais obtidos a partir de s_t e de s_d serão denotados respectivamente por s_l e s'_l.

O índice s_l pode ser calculado em tempo $O(|x||y|)$ e espaço $O(\min\{|x|,|y|\})$ por um esquema simples de programação dinâmica. De fato, ponha $M[0,j] = M[i,0] = 0$ para $0 \leq i \leq |x|, 0 \leq j \leq |y|$ e $M[i,j] = M[i-1,j-1] + s(x_i, y_j)$ se essa soma for maior ou igual a zero, ou 0 caso contrário para $1 \leq i \leq |x|, 1 \leq j \leq |y|$. Prova-se então trivialmente que $s_l(x,y)$ é igual ao valor máximo das entradas de M. O cálculo de s'_l pela definição através do algoritmo de Needleman-Wunsch consumiria tempo $O(|x|^2|y|^2)$ e espaço $O(\min\{|x|,|y|\})$ (isso corresponde a executar o algoritmo para cada par $(u,v) \in \mathrm{Suf}(x) \times \mathrm{Suf}(y)$, determinando o valor máximo atingido pelas entradas de todas as matrizes obtidas).

Exceto no caso de s_t, para todos os índices que demos, locais ou não, o desempenho dos algoritmos conhecidos para calculá-los (quadráticos ou piores) é proibitivo em se tratando de seqüências biológicas, e por isso via

[5] No caso de proteínas, usa-se com freqüência as matrizes PAM.

de regra esses índices não são calculados, e sim estimados. Como cada um desses índices, por definição, é o valor máximo de uma determinada função, a concessão básica que se faz é restringir heurísticamente o domínio dessa função, com o subseqüente cálculo do seu máximo nesse novo domínio. Isso nos leva, é claro, a estimativas inferiores.

3.1.3 FASTA

O programa FASTA [23] surgiu em 1988, e é um aperfeiçoamento do FASTP de 1985. A fim de enquadrá-lo dentro do ponto de vista que desenvolvemos anteriormente, podemos dizer, de forma grosseira, que esse programa obtém uma estimativa inferior do índice s'_l. É mais correto entretanto dizer que ele é um programa que deteta (ou busca) homologias.

Descreveremos rapidamente o FASTP [18], o que já será suficiente para ilustrar o tipo de heurística utilizada nessa família de programas que inclui também o LFASTA. A restrição do domínio em que se buscarão homologias baseia-se no *método diagonal*.

Dadas seqüências x e y e um inteiro $k > 0$, as *k-casamentos* entre x e y são os pares $(i, j) \in \{1, 2, \ldots, |x| - k + 1\} \times \{1, 2, \ldots, |y| - k + 1\}$ que satisfazem $x_{i+t} = y_{i+t}, t = 0, 1, \ldots, k - 1$. Os k-casamentos constituem a informação primária utilizada pela FASTP. O papel da constante k é regular a seletividade, a sensibilidade e a velocidade da busca. De modo geral, valores altos para k levam a buscas mais rápidas. Ao aumentarmos k, por um lado corremos o risco de deixar de detetar similaridades importantes (perda de sensibilidade), mas por outro deixa-se de considerar similaridades casuais sem significado biológico (ganho de seletividade).

Determinados os k-casamentos, passa-se a classificá-los pelas diagonais a que pertencem (o par (i, j) pertence à diagonal $i - j$). Nesse ponto, o modo mais simples de se restringir o domínio em que buscaremos similaridades é trabalhar-se apenas com as diagonais que contiverem relativamente maior número de k-casamentos. FASTP isola dentro das diagonais regiões que concentram grande número de k-casamentos, e seleciona cinco dessas regiões.

Cada uma das cinco regiões de alta similaridade selecionadas pareia um fator u de x com um fator v de y. O índice $s_t(u, v)$, com s dada pela matriz PAM250, é então calculado para os cinco pares de fatores. O maior dos cinco valores é tomado como *pontuação inicial* entre x e y, e a região associada recebe o nome de *região inicial*.

Nesse ponto é necessário ressaltar que o FASTP é um programa do tipo *um-contra-todos*. A partir de uma seqüência-argumento x e um banco, a pontuação inicial entre x e cada seqüência do banco é calculada. A média e o desvio padrão dessas pontuações é obtida e para cada seqüência do banco cuja pontuação for significativa em vista desses dois parâmetros, uma nova pontuação é calculada.

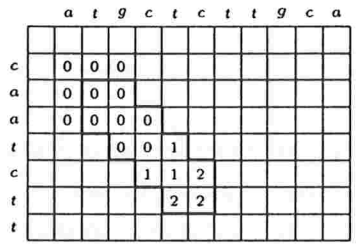


Figura 3.4: Algoritmo de Needleman-Wunsch restrito; a função s é como na figura 3.3.

Essa nova pontuação considera, de forma restrita, inserções e deleções de caracteres. Trata-se de aplicar o algoritmo de Needleman-Wunsch numa *faixa de largura fixa* em torno da região inicial. De modo geral, A matriz calculada pelo algoritmo de Needleman-Wusnch pode ser trivialmente restringida (na figura 3.4 damos um exemplo). A saída do programa consiste nessas novas pontuações e dos alinhamentos associados, além de um histograma de número de seqüências contra pontuações iniciais.

3.1.4 BLAST

O programa BLAST [1], surgido em 1990, obtém uma estimativa inferior do índice s_l. Os autores do BLAST chamam tanto $s_l(x, y)$ quanto um par $(u, v) \in \text{Fat}(x) \times \text{Fat}(y)$ que satisfaça $s_t(u, v) = s_l(x, y)$ de *MSP* (Maximal Segment Pair). Ao contrário do FASTP, a heurística empregado para a obtenção da estimativa fundamenta-se, ao menos parcialmente, em alguns estudos probabilísticos teóricos.

Ao buscar homologias entre duas seqüências, BLAST necessita preprocessar uma delas. Assim como FASTA, BLAST busca homologias entre uma

seqüência-argumento x e todas as seqüências de um banco. Assim, a escolha natural para a seqüência a ser preprocessada é x.

```
CTTTCTCGCAGGGAAATCTCGAATTTTCCCCTCCCGGCGACA|GAGT|ATAAATACGGGCG
|||      || |        |  ||  |       ||| |  |||| |||||||  ||||
CTTCGAGAGAGCGCGCCTCGAATGTTCGCGAAAAGAGCGCCG|GAGT|ATAAATAGAGGCG
```

Figura 3.5: Alinhamento produzido pelo BLAST.

O primeiro passo consiste em calcular uma lista $L = L(x)$ de palavras que, no caso de ácidos nucleicos, consiste de todos os fatores de tamanho 8 (por exemplo) de x, e no caso de proteínas, consiste de todas as palavras z de comprimento 8 (por exemplo) para as quais existe um fator u de x tal que $s_t(u, z)$ supera um determinado valor de corte estabelecido experimentalmente. A função s neste caso é dada pela matriz PAM120. No caso de proteínas, o tamanho de L, sob determinadas condições, gira em torno de $50|x|$.

O segundo passo é, lida uma seqüência y do banco, determinar todas as ocorrências de elementos de L em y (para isso usa-se uma estratégia baseada em autômatos finitos) e tentar extendê-las para pares $(u, v) \in \mathrm{Fat}(x) \times \mathrm{Fat}(y)$ para os quais $s_t(u, v)$ seja significativamente alto.

Suponha que $y_j y_{j+1} \ldots y_{j+7} = z$ onde z é um elemento de L. Seja $i\{1, 2, \ldots, |x| - 7\}$ tal que $x_i x_{i+1} \ldots x_{i+7} = z$ (no caso de ácidos nucleicos) ou $s_t(x_i x_{i+1} \ldots x_{i+7}, z)$ supera o valor de corte (no caso de proteínas). Agora para cada $k = 0, 1, 2, \ldots$ - nessa ordem - calcula-se $s_t(x_{i-k} x_{i-k+1} \ldots x_{i+7}$, $y_{j-k} y_{j-k+1} \ldots y_{j+7}$ (supomos $i - k, j - k > 0$) até que esse índice caia, por exemplo, 20 unidades abaixo do maior valor que ele tiver atingido para os valores anteriores de k. Note que o cálculo de cada um desses índices (exceto talvez o primeiro) consome nada mais que uma soma. Uma extensão análoga para a direita é feita em seguida.

Consideradas as duas extensões, devemos decidir se o maior valor de s_t obtido é ou não significativo. Para isso conta-se com resultados teóricos que estimam, dado S, a probabilidade de que duas seqüências aleatórias de comprimentos $|x|$ e $|y|$ contenham fatores u e v (u é fator de uma das palavras, v da outra), satisfazendo $s_t(u, v) > S$. Com isso pode-se estabelecer

de forma natural um limite inferior que, sempre que superado pelo maior valor de s_t obtido durante as extensões, permita-nos considerar significativo o pareamento de fatores obtido. Os fatores e o valor de s_t são então mandados para a saída.

Pode-se encarar o BLAST como uma ferramenta a partir da qual é possível, através de composições, construir outras. É esse o ponto de vista dos seus autores, expressado no próprio nome *Basic Search Tool*. Há, por exemplo, um programa que implementa o mesmo algoritmo utilizado pelo BLAST, com alterações mínimas, para classificar os fatores de uma família de seqüências de DNA em *clusters* [22]. Um outro exemplo é a estratégia para processar a saída do BLAST descrita em [21]. Essa saída por vezes é excessivamente volumosa, e a estratégia procura compor os pareamentos, ao mesmo tempo que elimina alguns deles. Com isso apresenta-se para o especialista dados mais legíveis.

3.2 Busca de padrões repetidos

Longas palavras sobre um alfabeto finito apresentam inevitavelmente padrões repetidos. Em cadeias de DNA, entretanto, repetições exatas ou aproximadas ocorrem acima do esperado em seqüências aleatórias. Muitas vezes essas repetições têm um papel biológico bem definido.

A detecção de repetições exatas não exige um grande esforço computacional; as estruturas baseadas em sufixos que já apresentamos resolvem esse problema eficientemente. Apresentaremos uma técnica baseada em análise de transformadas, que permite determinar, com algumas restrições, repetições aproximadas de forma eficiente.

3.2.1 Repetições em cadeias de DNA

A ocorrência de padrões repetidos em cadeias de DNA não é um fenômeno raro. Por exemplo, há genes que ocorrem várias centenas de vezes no DNA humano. Quanto maior for o número de cópias de um mesmo gene, maior será a freqüência com que a proteína associada será sintetizada. Um outro exemplo são as seqüências ALU. Cada uma delas tem cerca de trezentas bases. Apresentam um alto índice de similaridade se consideradas duas a duas, e perto de um milhão delas ocorrem no DNA humano. Nesses dois casos

a detecção das repetições extrapola os limites dos métodos computacionais, que podem ser empregados, entretanto, em contextos locais, como no interior de um gene.

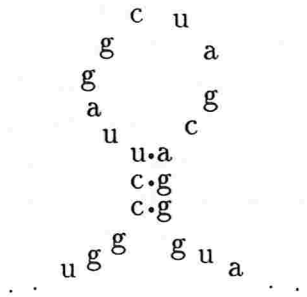


Figura 3.6: As simetrias fazem com que o RNA forme pareamentos, determinando dessa forma parte da estrutura espacial da molécula.

As *simetrias* são uma variante importante de padrões repetidos que ocorrem em cadeias de DNA. O *complementar* de uma seqüência de bases é a seqüência que se obtém revertendo-a e trocando-se A por T e C por G (e vice-versa). Por exemplo, o complementar de $AACTGA$ é $TCAGTT$. A complementaridade está associada com o fenômeno do *pareamento de bases*: bases complementares ligam-se quimicamente, o que faz com que cadeias complementares liguem-se base a base dando origem a cadeias duplas. Isso permite a duplicação da informação genética, que pode dessa forma ser transmitida aos descendentes, e também a cópia de trechos dela, por exemplo no processo de síntese proteica. Esse fenômeno facilita ainda a localização de genes no DNA.

Uma *simetria* é a ocorrência simultânea de uma seqüência e do seu complementar numa cadeia de DNA. Num gene, *simetrias* são importantes na determinação da estrutura espacial do *ácido ribonucleico mensageiro mRNA*, elemento intermediário no processo de síntese proteica. O reconhecimento do término de um gene dentro de uma cadeia de DNA também envolve a ocorrência de uma *simetria*.

Repetições exatas, incluindo *simetria* podem ser facilmente detetadas através de estruturas baseadas em sufixos. Já tivemos a oportunidade de dizer que a referência mais antiga que conhecemos do vetor dos sufixos é uma a-

plicação sua ao problema da detecção de padrões repetidos em cadeias de DNA.

Pode-se pensar em empregar estruturas baseadas em sufixos também na busca de repetições aproximadas. Uma repetição aproximada é composta por várias repetições exatas, e é nisso que se baseia, por exemplo, a métrica [6] de seqüências computável em tempo linear de Ehrenfeucht e Haussler [10].

Qualquer técnica de detecção de similaridades locais entre duas seqüências pode em princípio ser adaptada para determinar repetição aproximada de padrões, uma vez que esses dois problemas são de certa forma o mesmo problema, e por isso aplica-se aqui o que já dissemos sobre detecção de homologias. A técnica que apresentaremos a seguir, entretanto, baseia-se na transformada discreta de Fourier.

3.2.2 Análise de transformadas

A transformada discreta de Fourier vem sendo utilizada para detetar padrões repetidos em seqüências de DNA, de forma semelhante àquela em que é utilizada para detetar componentes periódicas fortes em análise de séries temporais. A vantagem desse método é que ele permite lidar com repetições aproximadas, e pode ser levada a cabo com eficiência. É notório entretanto o seu caráter heurístico, não havendo, até onde sabemos, estudos cuidadosos dos seus limites de aplicação.

Fixado um inteiro $n > 0$, os vetores $g^j = (e^0, e^{-i(j-1)2\pi/n}, e^{-2i(j-1)2\pi/n}, \ldots, e^{-(n-1)i(j-1)2\pi/n})$ onde j varia de 1 a n formam uma base ortogonal de $\mathfrak{C}^n$, onde $\mathfrak{C}$ é o corpo dos complexos e $i = \sqrt{-1}$ é a unidade imaginária. Calcular a transformada discreta de Fourier de $f \in \mathfrak{C}^n$ significa determinar a representação de f nessa base, isto é, determinar $h \in \mathfrak{C}^n$ que satisfaz $f = \sum_{j=1}^{n} h_j g^j$. Note que da base ser ortogonal e de que $g^j g^j = n$ $(1 \leq i \leq n)$ vem que $h_j = fg^j/n$, onde fg^j é o produto escalar de f por g^j. Lembramos que o vetor h pode ser calculado em tempo $O(n \log n)$ se usarmos algum esquema rápido (FFT's).

Seja x uma palavra sobre $\{a, c, t, g\}$ de comprimento n. Associaremos a x os vetores característicos f^a, f^c, f^t e f^g dados por $f^a_j = 1$ se $x_j = a$, ou 0 caso contrário, etc. Sejam h^a, h^c, h^t e h^g as transformadas discretas de Fourier

[6] Essa métrica é definida da seguinte forma: dadas seqüências u e v, seja $\delta_v(u) = 0$ se u for fator de v, ou $\delta_v(u(i, |u|))$ onde $u(1, i-1)$ é o menor prefixo de u que não é fator de v. Nessas condições, $(u, v) \mapsto \log((\delta_v(u) + 1)(\delta_u(v) + 1))$ é uma métrica.

desses vetores característicos. Seja $h_j = h_j^a\overline{h_j^a} + h_j^c\overline{h_j^c} + h_j^t\overline{h_j^t} + h_j^g\overline{h_j^g}$, onde a barra indica conjugação complexa. Então h é um vetor de números reais e podemos plotar as suas entradas num gráfico $h_j \times j$.

Fixado j, a periodicidade de $m \mapsto e^{-mi(j-1)2\pi/n}$ acarreta uma periodicidade nos vetores da base. Por exemplo, se $j > 1$ então admitindo por simplicidade que $T = n/(j-1)$ é um número inteiro, temos $g_t^j = g_{t+T}^j$, todo $1 \leq t \leq n - T$. Assim, a decomposição de f nessa base é uma decomposição de f em *componentes periódicas*. Note que os vetores da base têm norma euclidiana igual a $\sqrt{n}$, e por isso para comparar as normas das componentes (a fim de identificar as mais significativas), é suficiente comparar as normas dos coeficientes.

Vejamos agora a relação entre o vetor h e as repetições de padrões em x. Para cada inteiro $k > 0$ definimos o número de *k-coincidências* de x como sendo o número de índices $i \in \{1, 2, \ldots, |x| - k\}$ que satisfazem $x_i = x_{i+k}$. Ao menos em alguns casos particulares, podemos verificar diretamente que um valor alto para o número de k-coincidências acarreta um valor alto para h_j, onde $j \approx 1 + n/k$.

Por exemplo, se $x = acac$, então a base é $g^1 = (1,1,1,1), g^2 = (1,i,-1,-i)$, $g^3 = (1,-1,1,-1)$ e $g^4 = (1,-i,-1,i)$, e os vetores característicos são $h^a = (1,0,1,0)$ e $h^c = (0,1,0,1)$. Os números de k-coincidências são $0, 2$ e 0 para $k = 1, 2, 3$. Note que $h^a g^2 = h^c g^2 = h^a g^4 = h^c g^4 = 0$, enquanto $h^a g^3 = 2$ e $h^c g^3 = -2$. Um rápido exame revela que isso se deve ao fato das entradas de g^3 repetirem-se com período 2, enquanto cada duas entradas de g^1 (ou g^4) que distem duas unidades entre si anulam-se quando somadas. O vetor h é $(1/2, 0, 1/2, 0)$.

Não sabemos em geral se é possível, a partir de h e j, determinar um intervalo não trivial que contenha, por exemplo, o número de $\lfloor n/(j-1) \rfloor$-coincidências de x. O que podemos fazer é gerar seqüências com um número relativamente alto de k-coincidências, provocada pela presença de duas ou mais cópias equiespaçadas de um mesmo padrão, e em seguida obter o gráfico de $h_j \times j$ associado, procurando nele picos nas abscissas relativas às componentes de periodicidade k.

É isso basicamente o que fazem os trabalhos que consultamos [25, 27]. A diferença dos nossos testes é que trabalhamos com seqüências artificiais, ao invés de seqüências biológicas. Com isso pretendemos testar experimentalmente a eficácia desse método, fazendo variar os parâmetros que controlam a repetição de padrões nas seqüências que geramos.

Nosso programa gera aleatoriamente uma seqüência de tamanho n sobre m letras, e em seguida *carimba* um padrão u em várias posições de x. *Carimbar* u na posição i de x significa transformar x trocando x_{i+j-1} por u_j, $j = 1, 2, \ldots, |u|$. As posições em que carimbamos padrões são equiespaçadas, isto é, da forma $i+T, i+2T, i+3T, \ldots$. Para que a repetição de padrões não possa ser trivialmente detetada com alguma estrutura baseada em sufixos, não carimbamos os padrões de forma exata, mas com 10% de erros (caracteres trocados) Por exemplo, a seqüência caccaaaccccccccacacccccaccccaa foi produzida pelo nosso gerador. O padrão *accc* ocorre quatro vezes nela de forma equiespaçada, com dois erros.

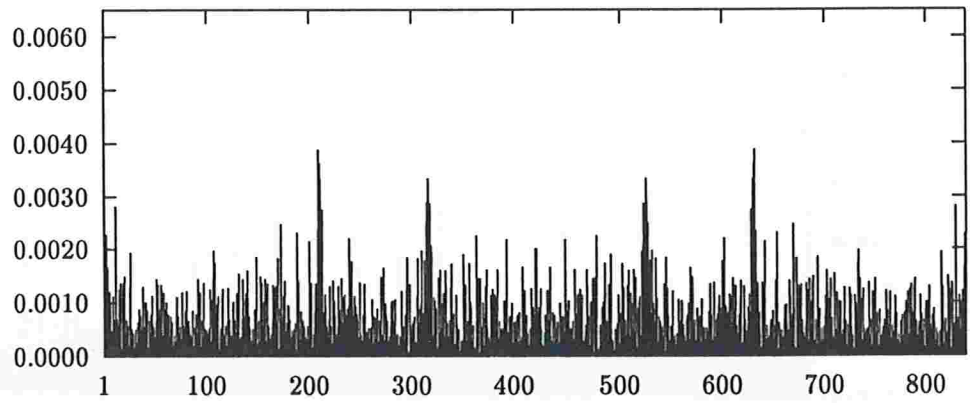


Figura 3.7: O comprimento da seqüência é 839, e o do padrão 8. O número de ocorrências é 16, e o espaçamento é o mínimo possível (8 unidades).

Geramos cem seqüências sobre quatro letras, com dois comprimentos possíveis e carimbadas de várias formas distintas. Trabalhamos com os comprimentos 839 e 1024, carimbando padrões de comprimento 1% do comprimento da seqüência gerados aleatoriamente. O critério para a escolha dos comprimentos foi simplesmente tomar um que fosse primo e outro potência de dois. Os demais parâmetros foram escolhidos arbitrariamente. O parâmetro que nos permitirá dar uma visão unificada dos resultados dessas experiências é a *cobertura*. Se um padrão u é carimbado r vezes em x, então diremos que *a cobertura é de* $(100r|u|/|x|)\%$.

Submetemos cada seqüência gerada e carimbada à análise de Fourier descrita, procurando com isso detetar a repetição que sabíamos existir. O critério que adotamos para decidir se a repetição é detetável ou não foi visual: olhamos o gráfico de $h_j \times j$ procurando nele picos claramente distinguíveis. Não aplicamos qualquer teste estatístico como o de Fischer[7].

Para os comprimentos de seqüência com que trabalhamos, a uma cobertura de 10% via de regra pôde-se distinguir alguns picos. A 15% eles tornaram-se mais nítidos. As experiências sugerem que quanto menor o espaçamento, mais distinguíveis são os picos (deve-se notar também que à medida em que aumentamos o espaçamento, multiplicamos os picos – veja os exemplos a seguir).

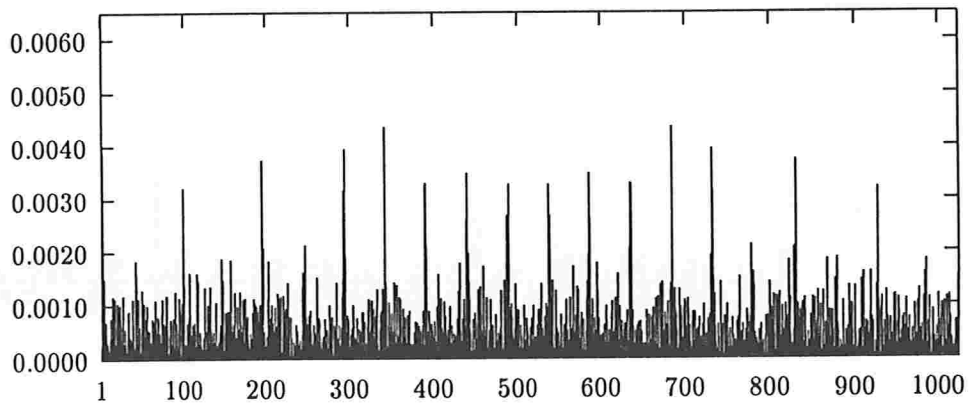


Figura 3.8: O comprimento da seqüência é 1024, e o do padrão 10. O número de ocorrências é 20, e o espaçamento é 21.

Apresentamos dois exemplos. No primeiro (figura 3.7), o comprimento da seqüência é 839, e a cobertura é de 15% (dezesseis ocorrências de um padrão de comprimento 8). O espaçamento é de 8 unidades. Distinguem-se picos próximos às abscissas 200 e 300. Note que $1 + 2(839/8) \approx 211$ e $1 + 3(839/8) \approx 316$. Os picos da metade direita do gráfico podem ser desconsiderados, pois da definição da transformada deduz-se facilmente que

[7]Os cálculos foram feitos com o programa RLaB de Ian Searle, e para gerar os gráficos usamos o programa Gnuplot, de Colin Kelley e Thomas Williams.

para $1 < j \leq n/2$ vale $h_j = h_{n-j+2}$. Os gráficos de várias seqüências – não todas – geradas com os mesmos parâmetros apresentaram esse mesmo aspecto.

No segundo exemplo (figura 3.8) temos um espaçamento relativamente maior (duas vezes o comprimento do padrão), mas em compensação aumentamos a cobertura para 20%. Há diversos picos próximos aos múltiplos inteiros de 50 (note que $1024/21 \approx 49$).

Quanto à existência (ou não) de um valor mínimo aproximado para que uma cobertura possa ser detetada através dessa técnica, deve-se ressaltar que a probabilidade de, numa seqüência randômica de comprimento n sobre quatro letras, encontrarmos um padrão de comprimento $n/100$ repetido 10 vezes, tende a zero à medida que n cresce (veja [17]). Assim, se a cobertura mínima não decrescer com n, então a técnica perde sensibilidade com grande rapidez à medida que n aumenta.

3.3 Contagem de fatores

A distribuição das repetições exatas de fatores é um meio de distinguir estruturalmente duas seqüências (veja por exemplo [17]). Grandes frações do DNA não tem um papel biológico notável, e elas distinguem-se dos genes por características que podem ser detetadas através de contagem de fatores repetidos.

3.3.1 Regiões codificadoras

Os genes somam apenas uma pequena fração dispersa do DNA. Ao seqüenciar-se um fragmento de DNA, pode-se não saber de antemão se ele faz parte ou não de um gene. O início e o término de um gene são reconhecidos pela enzima sintetizadora de mRNA, e poderia-se pensar em construir um teste computacional baseado nos mecanismos químicos de reconhecimento dessas posições especiais.

De fato, conhece-se algumas características estruturais peculiares aos inícios e términos de genes. Por exemplo, no trecho de DNA que antecede um gene costumam ocorrer de forma aproximada os padrões $TTGACA$ e $TATAAT$ nas bactérias e o padrão $TATAAA$ nos eucariotas, além de, em alguns casos, padrões palíndromes (palavras iguais aos seus complementa-

res). Quanto aos términos, já dissemos que uma das suas características é a presença de *simetrias.*

Outra abordagem possível fundamenta-se na constatação de que os genes, enquanto seqüências de letras, exibem algumas características estruturais que os distinguem do restante do DNA. Uma delas é a chamada *preferência de códons.* Além disso, alguns padrões pequenos aparecem em genes com uma freqüência significativamente superior que em outros trechos de DNA.

```
DROHSP22 agagtgccggtattttCTAGATTATATGGAtttcctctctgtcaagagTATAAAtagccac
DROHSP26 ccttttctgtcactttCCGGACTCTTCTAGaaaagctccagcgggTATAAAagcagcgtc
DROHSP23 cacacacagcgccgacgggcgccgcacacttcgacagcaagcggttgTATAAAtatccggc
DROHSP27 ttccgtccctggttgccatgcaCTAGTGTGTGTGAGCCcagcgtcagTATAAAagccggcg
DROHSP70 gcttcgagagagcgcgcCTCGAATGTTCGCGaaaagagcgccggagTATAAAtagaggcgc
DROHSP83 gcctctaggagtttCTAGAGACTTCCAGttcgggtgcgggtttttcTATAAAagcagacgc
DROHSP68 ccctttctcgcagggaaatCTCGAATTTTCCCCtcccggcgacagagTATAAAtacgggcg
```

Figura 3.9: O padrão *tataaa* e a palíndrome *ctcgaatattcgag* ocorrendo de forma aproximada nas regiões que antecedem alguns genes da *Drosophila*, com as identificações do GenBank (figura retirada de [29]).

Tanto num caso como no outro, as características estruturais próprias dos genes podem ser detetadas por um estudo dos seus *fatores repetidos.* Como essas repetições são exatas, podemos lançar mão de estruturas baseadas em sufixos, que é o que faremos a seguir de forma unificada, através da *Complexidade de Trifonov.*

3.3.2 Complexidade

A *Complexidade de Trifonov* [28] associa a cada palavra um coeficiente entre 0 e 1. Ela é um bom exemplo do uso do autômato dos sufixos em contagem de fatores. Intuitivamente, uma palavra tem *baixa complexidade* quando ela admite uma descrição natural simples, como por exemplo a potência $(ag)^4 = agagagag$. Tais palavras deverão receber coeficientes próximos de 0. Por sua vez, uma palavra cuja descrição natural mais simples é ela mesma (por exemplo *aagccacgga*), têm *alta complexidade* e deverá receber um coeficiente próximos de 1.

A formalização da noção de complexidade apóia-se numa outra, a saber, o *uso de fatores*. Considere que o alfabeto seja $\{a, c, g\}$. Na palavra $(ag)^5$, ag ocorre cinco vezes e ga quatro, enquanto gg ou ac por exemplo não ocorrem nenhuma vez. Por outro lado, na palavra *aagccacgga* cada palavra em $\{a, c, g\}^2$ ocorre exatamente uma vez. No segundo caso, diremos que houve um *melhor uso* dos fatores de comprimento dois.

De maneira geral, seja $k > 0$ um inteiro menor ou igual a $|x|$. Suponha por um momento que $\mu = \mu(x, k) = (|x|-k+1)/|A^k|$ seja um número inteiro. Sempre que $u \in A^k$ ocorrer mais do que μ vezes em x, o *excesso* $|t_x(u)| - \mu$ se refletirá em que outras palavras de A^k não poderão ocorrer μ vezes em x, e por isso diremos que esse excesso é o *desperdício de u* (quando $|t_x(u)| \leq \mu$ o desperdício de u é zero). Posto isso, considere a diferença entre $|x| - k + 1$ e a soma dos desperdícios de todas as palavras de A^k. O *uso* $U_k = U_k(x)$ *que x faz dos fatores de tamanho k* é então definido como o quociente dessa diferença por $|x| - k + 1$. O efeito dessa divisão é normalizador.

Quando μ não for inteiro bastará definir r como o resto da divisão de $|x|-k+1$ por A^k e, para r palavras $u \in A^k$, considerar que o desperdício é $|t_x(u)|-\lceil\mu\rceil$. Para as outras palavras u de A^k consideraremos que o desperdício é $|t_x(u)| - \lfloor\mu\rfloor$. A escolha dessas r palavras deverá ser feita de modo a maximizar o uso U_k. Não é difícil ver que se k for maior que o comprimento do fator repetido mais longo de x então $U_k = 1$ (em particular $U_{|x|} = 1$).

Vejamos dois exemplos. Se $x = a^n$, então para $1 \leq k < n$ temos $U_k = \lceil\mu(x,k)\rceil/(n-k+1) = \lceil(n-k+1)/|A^k|\rceil/(n-k+1) \leq 1/|A^k|+1/(n-k+1)$. Tomemos agora para x uma palavra de Good-de Bruijn sobre A a parâmetro r. Se $k \geq r$ então $U_k = 1$. Para $k < r$, seja $u \in A^k$. Então u ocorre pelo menos $|A^{r-k}|$ vezes em x, pois há $|A^{r-k}|$ modos diferentes de completar u (concatenando letras à direita) para um elemento de $|A^r|$. Como $\mu(x,k) \geq |A^{r-k}|$, temos que $|A^r|/(|A^r| - k + 1)$ é uma estimativa inferior para U_k.

Não é difícil descrever um algoritmo para o cálculo de U_k. Seja f_x a função que associa a cada palavra sobre A o seu número de ocorrências em x. Seja $u_1, u_2, \ldots, u_n$ uma enumeração de A^k que disponha os elementos de A^k em ordem decrescente do número de ocorrências em x, ou seja, $f_x(u_i) \geq f_x(u_j)$ sempre que $1 \leq i \leq j \leq n$. Dado $i \in \{1, 2, \ldots, n\}$, definimos δ_i como sendo 1 se $i \leq r$ ou 0 caso contrário. Então o uso que x faz dos fatores de comprimento k é $\sum_{i=1}^{n} \min\{f_x(u_i), \lfloor\mu\rfloor + \delta_i\}$.

A parte delicada desse processo é a obtenção da seqüência $|t_x(u_1)|, |t_x(u_2)|, \ldots, |t_x(u_n)|$. Para isso usaremos – sem grande sofisticação –, o autômato dos

sufixos da forma que segue. Calcularemos, para cada estado α do autômato dos sufixos $\mathcal{A}$ de x, o número $|S_{\mathcal{A}}(\alpha)|$ de caminhos que partem de α e atingem algum estado final. Isso pode ser feito ao longo da construção de $\mathcal{A}$ (sem prejuízo para o tempo de execução) ou após o seu término. Nos dois casos o procedimento a seguir é elementar.

```
1   função Complexidade(x)
2   construa A
3   inicialize L_1, L_2, ..., L_m como listas vazias
4   para cada α ∈ Q_A faça
5     para i de prof[α] − |P_A(α)| + 1 até min{m, prof[α]} faça
6       adicione |S_A(α)| a L_i
7   c ← 1
8   para k de 1 até m faça
9     ordene L_k em ordem decrescente
10    u ← 0; μ ← (|x| − k + 1)/|A^k|; r ← |x| − k + 1 mod |A|^k
11    enquanto L_k for não vazia faça
12      remova o primeiro elemento a de L_k
13      u ← u + min{⌊μ⌋ + s(r), a}
14      r ← r − 1
15    c ← c * u/|A^k|
16  devolva(c)
```

Algoritmo 3.1: Cálculo da Complexidade de Trifonov através do autômato dos sufixos. O inteiro m é o comprimento do fator repetido mais longo de u, que pode ser trivialmente obtido ao longo da construção do autômato dos sufixos.

Dado um estado α de $\mathcal{A}$, o número $|P_{\mathcal{A}}(\alpha)|$ de caminhos que partem de ι e atingem α é 1 se $\alpha = \iota$ ou $prof[\alpha] - prof[pai[\alpha]]$ caso contrário, e portanto o cálculo de $|P_{\mathcal{A}}(\alpha)|$ pode ser feito em tempo constante. Note que os comprimentos desses caminhos são $prof[\alpha], prof[\alpha]-1, \ldots, prof[\alpha]-|P_{\mathcal{A}}(\alpha)|$.

Considere que a lista de inteiros L_i é inicialmente vazia, onde i varia de 1 até $|x|$. Para cada estado α de $\mathcal{A}$, adicione $|S_{\mathcal{A}}(\alpha)|$ à lista L_i, todo $i \in \{prof[\alpha], prof[\alpha]-1, \ldots, prof[\alpha]-|P_{\mathcal{A}}(\alpha)|\}$. Ao final, ordene na ordem

decrescente cada uma das listas. Feito isso, a lista L_k ao trecho inicial da seqüência $|t_x(u_1)|, |t_x(u_2)|, \ldots, |t_x(u_n)|$ formado pelos seus elementos não nulos. De fato, para cada fator $u \in \mathrm{Fat}(x)$ de comprimento k o procedimento que descrevemos adiciona a L_k o número $|t_x(u)|$ uma e apenas uma vez (fá-lo ao visitar $(\iota u)_{\mathcal{A}}$).

O raciocínio que acabamos de fazer mostra ainda que soma dos comprimentos das listas construídas é $|\mathrm{Fat}(x)|$ e, portanto quadrática em $|x|$. O tempo total de construção das listas é claramente limitado por essa soma. O fato dela ser quadrática não chega a preocupar, pois poderemos considerar o compimento de x limitado (e pequeno), como veremos a seguir.

fragmento	posições	complexidade
exon 1	3437 – 3478	0.58
intron A	3479 – 3980	0.37
exon 2	3981 – 4157	0.52
intron B	4158 – 4959	0.46
exon 3	4960 – 5100	0.58
intron C	5101 – 5919	0.36
exon 4	5920 – 6058	0.54
intron D	6059 – 6389	0.39
exon 5	6390 – 6502	0.51
intron E	6503 – 7747	0.45
exon 6	7748 – 7981	0.49
intron F	7982 – 8268	0.48
exon 7	8269 – 8375	0.46
intron G	8376 – 9001	0.45
exon 8	9002 – 9098	0.57

Tabela 3.1: Complexidade média de alguns intros e exons (MUSGPD).

Estando definida a noção de uso de fatores, podemos voltar à complexidade. A *Complexidade de Trifonov* de x é o produto $U_1 U_2 \ldots U_{|x|}$. Como para k maior que o comprimento m do fator repetido mais longo de x vale $U_k = 1$, podemos truncar o produtório considerando apenas os m primeiros fatores.

Não se calcula a complexidade de longas seqüências biológicas, mas sim a *complexidade média*, que se define da forma seguinte: fixada uma constante

T (um valor típico é $T = 20$), a complexidade média de x (supomos $|x| \geq T$) é a média aritmética das complexidades de $x_i x_{i+1} \ldots x_{i+T-1}$ para i variando de 1 até $|x| - T + 1$, inclusive.

A complexidade média aparentemente é uma ferrramenta útil para o estudo das seqüências biológicas. Algumas experiências sugerem que a complexidade média de trechos de seqüências de DNA correspondentes a genes é maior que a complexidade média de trechos que não correspondem a genes, donde essa medida poderia ser utilizada para classificar trechos de DNA recém-sequenciados.

Implementamos esse algoritmo em linguagem C, inicialmente numa estação de trabalho Sun. Escolhemos alguns genes ao acaso no GenBank e calculamos a complexidade média de cada intron e de cada exon [8]. Neles, de modo geral a complexidade dos exons manteve-se abaixo da complexidade dos introns [9]. Escolhemos um deles (MUSGPD) para ser apresentado (tabela 3.1). A numeração das bases é a mesma dada no GenBank.

[8] Nem todos os genes correspondem a frações contínuas de DNA. Em organismos superiores, é comum que eles estejam fragmentados. Nesse caso, cada fragmento é chamado um *exon* desses genes. As porções de DNA que estão entre dois exons são chamadas *introns*.

[9] O próprio Trifonov deverá publicar um estudo estatístico cuidadoso em breve.

Índice

Bibliografia

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. A basic local alignement search tool. *Journal of Molecular Biology*, 1990.

[2] R. A. Baeza-Yates and G. H. Gonnet. All-against-all sequence matching (preliminary version). 1990.

[3] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstrings. In *STOC 91*, 1991.

[4] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M. T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[5] A. Blumer, A. Ehrenfeucht, and D. Haussler. Average sizes of suffix trees and dawgs. *Discrete Applied Mathematics*, 24:37–45, 1989.

[6] B. Clift, D. Haussler, R. McConnell, T. D. Schneider, and G. D. Stormo. Sequence landscapes. *Nucleic Acids Research*, 14:141–158, 1986.

[7] M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.

[8] M. Crochemore, T. Lecroq, A. Czumaj, L. Gasieniec, S. Jarominek, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Lecture Notes in Computer Science*, 1992.

[9] P. Cull and J. L. Holloway. Divide and conquer approximate string matching: when dynamic programming is not powerful enough. Technical Report 92-20-06, Oregon State University, Computer Science Department, 1992.

[10] A. Ehrenfeucht and D. Haussler. A new distance metric on strings computable in linear time. *Discrete Appl. Math.*, 20:191–203, 1988.

[11] G. H. Gonnet. Examples of PAT applied to the Oxford English Dictionary. Technical Report OED-87-02, UW Centre for the New Oxford English Dictionary, University of Waterloo, 1987.

[12] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. Lexicographical indices for text: inverted files vs. pat trees. Technical Report OED-91-01, UW Centre for the New Oxford English Dictionary, University of Waterloo, 1991.

[13] J. L. Holloway. An annotated bibliography of algorithms applicable to molecular biology. Technical Report 92-50-01, Oregon State University, Computer Science Department, 1992.

[14] B. C. Huang and M. A. Langston. Practical in-place merging. *Communications of the ACM*, 31:348–352, 1988.

[15] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching. *Lecture Notes in Computer Science*, 520:240–248, 1991.

[16] S. Karlin, G. Ghandour, F. Ost, and L. J. Korn. New approaches for computer analysis of nucleic acid sequences. *Proc. Natl. Acad. Sci. USA*, 80:5660–5664, 1983.

[17] S. Karlin, F. Ost, and B. E. Blaisdell. Patterns in DNA and amino acid sequences and their statistical significance. In M. S. Waterman, editor, *Mathematical Methods for DNA Sequences*, pages 133–156. CRC Press, 1988.

[18] D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227:1435–1441, 1985.

[19] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. Technical Report TR 89-14, University of Arizona, 1989.

[20] H. M. Martinez. An efficient method for finding repeats in molecular sequences. *Nucleic Acids Research*, 11:4629–4634, 1983.

[21] J. Meidanis. *Algorithms for problems in computational genetics.* PhD thesis, University of Wisconsin-Madison, 1992.

[22] R. Parsons, S. Brenner, and M. J. Bishop. Clustering cDNA sequences. *Comp. Appl. Biosc.*, 8:461–466, 1992.

[23] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci. USA*, 85:2444–2448, 1988.

[24] D. Sankoff and J. B. Kruskal, editors. *Time Warps, String edits, and Macromolecules: the theory and practice of sequence comparison.* Addison-Wesley, 1983.

[25] B. D. Silverman and R. Linsker. A measure of DNA periodicity. *Journal of Theoretical Biology*, 118:295–300, 1986.

[26] L. Stryer. *Biochemistry.* W. H. Freeman and Company, third edition, 1988.

[27] S. Tavaré and B. W. Giddings. Some statistical aspects of the primary structure of nucleotide sequences. In M. S. Waterman, editor, *Mathematical Methods for DNA Sequences*, pages 117–132. CRC Press, 1988.

[28] E. N. Trifonov. Making sense of the human genome. *Structure & Methods*, 1:69–77, 1990.

[29] M. S. Waterman. Consensus patterns in sequences. In M. S. Waterman, editor, *Mathematical Methods for DNA Sequences*, pages 93–115. CRC Press, 1988.