Tópicos em Algoritmos sobre Seqüências

Alair Pereira do Lago Imre Simon

Julho de 2003

Prefácio

A combinatória das palavras [Lot83] é área bastante rica e diversas aplicações a outras áreas da Matemática como teoria dos grupos [dL02], semigrupos [dL96, dLS01], autômatos e linguagens formais [RS97a, RS97b, RS97c, Pin86] têm se desenvolvido ao longo dos anos.

Outras aplicações que têm ganhado bastante importância recentemente incluem o desenvolvimento de algoritmos [CR94] que possibilitem a análise, processamento e extração de informações em seqüências cada vez mais compridas, sejam elas textos disponíveis na internet [cit, goo], bibliotecas digitais [Les97, NMWP99] ou seqüências genômicas [Gus97]. Face às dimensões cada vez maiores destas seqüências, algoritmos eficientes têm sido cada vez mais requisitados. Ocorre que a elaboração, e posterior análise, de algoritmos eficientes requer sempre um ferramental matemático mais sofisticado [CLR90] que o necessário à elaboração de algoritmos mais imediatos mas geralmente ineficientes. Ademais, poucos são os programadores que têm a devida educação ou interesse matemáticos necessários.

Um exemplo clássico e importante disto, é o bem estudado e fundamental problema da busca de uma palavra num texto. Enquanto que uma solução para o problema pode ser imediatamente construída como será visto na Seção 3.1, ao longo da história foram elaborados vários outros algoritmos bem mais eficientes como os propostos por Knuth, Morris e Pratt [KMP77] e por Boyer e Moore [BM77]. As demonstrações de sua corretude e eficiência, contudo, mostram-se bem menos imediatas.

O objetivo do corrente livro é o de exibir alguns outros problemas importantes dentro do estudo de seqüências, apresentar e discutir algoritmos[1] que resolvam estes problemas. No caso de alguns dos algoritmos mais sofisticados e importantes, pretendemos expor o ferramental combinató-

[1]Por vezes mais de um algoritmo será apresentado para o mesmo problema de forma a permitir comparações.

rio necessário e dar as devidas demonstrações matemáticas, infelizmente ainda pouco difundidas a um nível elementar, particularmente no Brasil. Serão vistas várias aplicações, em particular às seqüências biológicas.

Assim, no Capítulo 1 veremos uma rápida revisão do ferramental combinatório elementar a ser usado no livro. No Capítulo 2, veremos o primeiro resultado central a ser apreciado neste trabalho: a construção em tempo linear da árvore dos sufixos de uma seqüência. Algumas aplicações para soluções eficientes de alguns problemas computacionais importantes serão então vistas no Capítulo 3, incluindo uma solução eficiente para a busca de uma palavra num texto e a procura de um mais comprido fator comum a duas seqüências. Já no Capítulo 4 veremos o segundo resultado central capaz de maravilhar quem escreve algoritmos de forma mais consciente: uma maneira de resolver o problema do menor ancestral comum (MAC) a dois vértices de uma árvore em tempo constante, independente do tamanho da mesma. É suposto um pré-processamento linear no tamanho da árvore. Então, no Capítulo 5 veremos uma aplicação conjunta dos dois resultados centrais. Será resolvido de forma eficiente um dos mais importantes e mais recorrentes problemas dentro da Biologia Molecular Computacional [MS97, Pev00], o da busca de padrão com erros.

Por fim, no Capítulo 6, veremos uma resenha de alguns algoritmos usados na comparação de seqüências, cujas idéias principais recorrem de diferentes formas numa das operações mais fundamentais dentro de Bioinformática, o alinhamento de seqüências. Estes algoritmos não são tão eficientes quanto os demais vistos até agora, o que impossibilita sua aplicação a seqüências muito compridas, como as do tamanho de um genoma completo. Por conta disto, várias heuristicas [AGM+90, TM99, AMS+97, PL88, Pea00] têm sido introduzidas de forma a que se possa eliminar estas limitações.

Nem todo problema admite uma solução eficiente, mas aqueles que admitem acabam por se tornar centrais. Tanto que novos modelos matemáticos e boas heurísticas para problemas antes intratáveis sempre acabam por recorrer a estes problemas, na busca de soluções eficientes ainda que às vezes aproximadas. Num processo normal de desenvolvimento de algoritmos aplicados à Biologia, propõem-se modelos matemáticos o mais realistas possíveis, para uma posterior implementação dos algoritmos. De fato, muitas vezes a eficiência de técnicas como a construção da árvores dos sufixos e a obtenção do menor ancestral comum interferem na elaboração destes modelos de forma que os algoritmos e heurísticas resultantes possam valer-se delas. Este é o caso de diversas implementações de alinhadores de seqüências [DKF+99, BPM+00] bem como ferramentas que se prestam ao estudo das repetições num seqüência genômica [KCO+01].

Os pré-requisitos a uma leitura mais apropriada do livro são bastante modestos e envolvem um curso de programação (um segundo curso é desejável, em particular se incluir uma exposição mínima sobre estruturas básicas de dados). Um melhor entendimento sobre alguns conceitos combinatórios elementares, como grafos e árvores, que serão rapidamente revistos no Capítulo 1 também ajudam. Também contribuem alguns conhecimentos de Análise como o conhecimento de limites para uma adequada compreensão da Análise de Algoritmos que por vezes será feita.

Material de apoio e eventuais erratas e versões futuras deste texto estarão disponíveis no sítio `http://www.ime.usp.br/~alair/TopAlgSeq.html`

Sumário

Lista de Algoritmos

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Neste capítulo introduziremos algumas definições de conceitos combinatórios amplamente utilizados em várias áreas da Combinatória bem como outras áreas da Matemática Discreta e da Ciência da Computação. Não temos aqui a pretensão de esmiuçar os detalhes que o assunto merece, mas apenas dar as definições básicas e elementares que serão necessárias ao desenvolvimento adequado do restante do livro. Alguns exemplos serão fornecidos, mas um leitor interessado deve fazer uso de literatura adequada [Lot83] caso queira aprofundar-se sobre os conceitos envolvidos.

Na Seção 1.1 vemos as definições comumente utilizadas na combinatória das palavras, que afinal permitem definir o objeto básico de nosso trabalho: as palavras (também chamadas seqüências). Na Seção 1.2 vemos as definições relativas à teoria dos grafos.

1.1 Combinatória das palavras

Seja A um *alfabeto finito*. Qualquer seqüência finita de letras de A é também chamada de *palavra em A*, ou simplesmente *palavra* se o alfabeto for claro. Outros sinônimos muitas vezes encontrados para as palavras incluem cadeias, strings e seqüências. Neste trabalho, preferiremos o termo palavra. Tipicamente, as letras serão representadas pelas primeiras letras do alfabeto a, b, c, ... enquanto que palavras serão representadas pelas últimas: w, x, y e outras. Também procuraremos manter a convenção matemática de reservar letras maiúsculas aos conjuntos e minúsculas aos elementos.

Vários alfabetos podem ser utilizados em situações práticas: o conjunto das 26 letras do alfabeto, o conjunto dos possíveis 256 valores de um byte, o conjunto dos 4 nucleotídeos $\{a, t, c, g\}$[1], o conjunto dos 20 aminoácidos que formam as proteínas, etc. Associadas a estes alfabetos, podemos citar, respectivamente, palavras como: uma palavra da língua portuguesa, um arquivo de computador, uma seqüência de DNA, uma proteína.

O *comprimento* de uma palavra w é o comprimento da seqüência finita e é denotado por $|w|$. O conjunto das palavras com letras em A (inclusive a palavra vazia 1)[2] é denotado por A^* e o conjunto $A^* \setminus \{1\}$ é denotado por A^+. Dado um inteiro qualquer $k \geq 0$, denotamos por A^k o conjunto das palavras de comprimento k com letras em A. Para $A = \{a, b, c, d\}$ temos os seguintes exemplos de palavras: *adbc*, *abcacb*, 1, *bca*. Seus respectivos comprimentos são 4, 6, 0 e 3.

A *concatenação* de duas palavras u e v é definida de forma natural e é denotada por $u \cdot v$, ou simplesmente uv. Observe que $|uv| = |u| + |v|$. É fácil verificar que trata-se de uma operação associativa, isto é $uvw = u \cdot v \cdot w = u \cdot (v \cdot w) = (u \cdot v) \cdot w$. O elemento neutro da concatenação é a palavra vazia 1. Assim, A^+ é um semigrupo[3] e A^* é um monóide[4]. Dado um inteiro qualquer $k \geq 0$ e uma palavra w, definimos w^k a potência de w a k como sendo a concatenação de k cópias de w. Dadas uma palavra w e conjuntos de palavras U e V, definimos

$$\begin{aligned} UV &\stackrel{\text{def}}{=} \{uv \mid u \in U,\ v \in V\}, \\ wV &\stackrel{\text{def}}{=} \{wv \mid v \in V\}, \\ Uw &\stackrel{\text{def}}{=} \{uw \mid u \in U\}. \end{aligned}$$

Dizemos $x \in A^*$ é *prefixo*, *sufixo* ou *fator* de $w \in A^*$ se $w \in xA^*$, $w \in A^*x$, ou $w \in A^*xA^*$, respectivamente. Um fator (prefixo, sufixo) de w é dito *próprio* se for distinto de w. Denotamos por $\mathrm{Pref}(w)$ o *conjunto dos prefixos de* w, por $\mathrm{Suf}(w)$ o *conjunto dos sufixos de* w e por $\mathrm{Fat}(w)$ o *conjunto dos fatores de* w. Dado $0 \leq k \leq |w|$, observe que há um único prefixo de w de comprimento k e um único sufixo de w de comprimento k.

[1]Neste trabalho, ao contrário da prática em Biologia, usaremos letras minúsculas para representar os nucleotídeos básicos das seqüências de DNA. Isto se deve ao interesse de manter a convenção matemática que reserva letras maiúsculas aos conjuntos e minúsculas aos elementos.

[2]É também comumente usado o símbolo λ para a palavra vazia.

[3]Conjunto munido de operação associativa.

[4]Semigrupo munido de elemento neutro.

A concatenação de palavras não é operação inversível[5], e portanto não se define a palavra w^{-1}. Contudo é bastante útil e freqüente o uso de notações e algumas das propriedades observadas sobre os grupos[6]. Sendo u um prefixo de w, denotamos por $u^{-1}w$ a palavra obtida pela remoção do prefixo u de w, ou seja, o sufixo de w de comprimento $|w| - |u|$. Analogamente, para v um sufixo de w, denotamos por wv^{-1} o prefixo de w de comprimento $|w| - |v|$. É imediato verificar que $w = u(u^{-1}w) = (wv^{-1})v$. Dados dois prefixos de w, o de menor comprimento é prefixo do de maior comprimento. Afirmativa análoga vale para sufixos. Sejam u um prefixo e v um sufixo de w tais que $|u| + |v| \leq |w|$. Como $u^{-1}w$ e v são sufixos de w e $|u^{-1}w| = |w| - |u| \geq |v|$, temos que v é sufixo de $u^{-1}w$ e é definida a palavra $(u^{-1}w)v^{-1}$. Analogamente, é definida $u^{-1}(wv^{-1})$. Como estas palavras são idênticas, usualmente denotamo-las simplesmente por $u^{-1}wv^{-1}$. Sejam inteiros $i = |u|+1$ e $j = |w|-|v|$. Denotamos ainda por $w[i\,..\,j]$ o fator $u^{-1}wv^{-1}$. Ademais, dizemos que i é uma *ocorrência do fator em* w. Denotamos $w[i\,..\,i]$ por $w[i]$, e $w[1]$ denota a primeira letra de w. Dadas uma palavra w e conjuntos de palavras U e V, estendemos esta notação para conjuntos de forma natural:

$$
\begin{aligned}
w^{-1}V &\stackrel{\text{def}}{=} \{w^{-1}v \mid v \in V \cap wA^*\},\\
Uw^{-1} &\stackrel{\text{def}}{=} \{uw^{-1} \mid u \in U \cap A^*w\}.\\
U^{-1}w &\stackrel{\text{def}}{=} \{u^{-1}w \mid u \in U,\ w \in uA^*\},\\
wV^{-1} &\stackrel{\text{def}}{=} \{wv^{-1} \mid v \in V,\ w \in A^*v\},\\
U^{-1}V &\stackrel{\text{def}}{=} \{u^{-1}v \mid u \in U,\ v \in V \cap uA^*\},\\
UV^{-1} &\stackrel{\text{def}}{=} \{uv^{-1} \mid u \in U \cap A^*v,\ v \in V\},
\end{aligned}
$$

Observe que $w(A^*)^{-1}$ é o conjunto $\mathrm{Pref}(w)$ dos prefixos de w, que $\mathrm{Suf}(w) = (A^*)^{-1}w$ e que $\mathrm{Fat}(w) = (A^*)^{-1}w(A^*)^{-1}$. Assim, $w = abcacb = abc \cdot acb = ab \cdot cac \cdot b$ implica que $abc = w[1\,..\,3]$, $acb = w[4\,..\,6]$ e $cac = w[3\,..\,5]$ são respectivamente prefixo, sufixo e fator de $abcacb$. Ademais, $b = w[2\,..\,2] = w[6\,..\,6]$ ocorre nas posições 2 e 6 em w.

[5]Para todo elemento y existe elemento inverso y^{-1} tal que $(xy)y^{-1} = x = y^{-1}(yx)$, para todo elemento x.

[6]Monóides em que todo elemento possui um inverso.

1.2 Grafos e Árvores

Um *grafo dirigido* $\mathbf{G}$, ou simplesmente *grafo*, é um par (V, E) onde V é um conjunto qualquer e $E \subseteq V \times V$ é um conjunto de pares ordenados de elementos de V. Cada elemento de E é chamado *aresta de* $\mathbf{G}$ e cada elemento de V é chamado *vértice de* $\mathbf{G}$. Dada a aresta $e = (u, v) \in E$, dizemos que $\alpha(e) = u$ é o *início de* u e que $\omega(e) = v$ é o *término de* e, respectivamente[7]. O início e o término de uma aresta são também chamados de *extremidades* da aresta. α e ω também são chamadas genericamente de *funções de incidência*. Quando não explicitado o conjunto de vértices V, estaremos supondo $V = \alpha(E) \cup \omega(E)$. Muitas vezes deixaremos implícito o conjunto de vértices, definindo o grafo a partir de seu conjunto de arestas. Tipicamente, vértices serão representados por letras minúsculas u, v e vizinhas enquanto que arestas ("edges") serão representadas por letras e, f, g, etc. No caso de vértices, esperamos que o contexto seja claro para evitar a confusão com a representação de palavras. Na Figura 1.1, por exemplo, temos um grafo com cinco vértices e dez arestas. A flecha numa aresta representa o término da mesma. Assim temos arestas de u para v e para x, por exemplo.

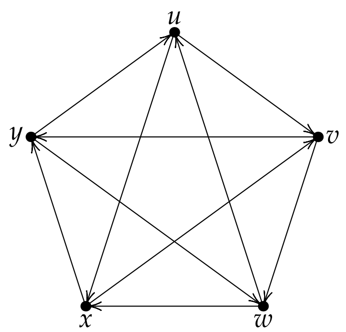


Figura 1.1: Exemplo de grafo

Seja $\mathbf{G} = (V, E)$ um grafo. Duas arestas $e, f \in E$ são ditas *consecutivas* (e nesta ordem) se $\omega(e) = \alpha(f)$. Chamamos de *passeio em* $\mathbf{G}$ ou simplesmente *passeio* toda seqüência finita de arestas consecutivas de $\mathbf{G}$. Sobre cada passeio $p = e_1 e_2 \cdots e_k$, $k \geq 1$ definimos o *início do passeio* $\alpha(p) = \alpha(e_1)$ e o *término*

[7]Muitas vezes relaxaremos esta terminologia de início e término e usaremos sinônimos tais como começo e fim. Também diremos que a aresta vai de um vértice para outro, parte de um vértice para outro, etc.

do passeio $\omega(p) = \omega(e_k)$. O início e o término[8] de uma passeio são também chamados de *extremidades* do passeio. Para cada vértice $v \in V$ definimos 1_v um *passeio vazio local a* v, com início e término em v. O *comprimento de um passeio* é o tamanho da seqüência de arestas, sendo que o comprimento dos passeios vazios é 0. Dizemos que dois passeios p, q são consecutivos (e nesta ordem) se $\omega(p) = \alpha(q)$. Dados dois passeios consecutivos p e q definimos sua concatenação pq de forma natural. Observe que pq começa em $\alpha(p)$ e termina em $\omega(q)$. Não é definida a concatenação de passeios não consecutivos. Observe que se p, q e r são passeios consecutivos nesta ordem, então $(pq)r = p(qr)$, que equivale a dizer que a operação parcial da concatenação de passeios é associativa quando todas as concatenações são definidas. No grafo da Figura 1.1, por exemplo, temos um passeio de comprimento 5 formado pelas arestas (w, u), (u, v), (v, y), (y, u), (u, x).

Dado um passeio $p = e_1 e_2 \cdots e_k$, com $k \geq 0$, dizemos que os vértices $\{\omega(e_1), \omega(e_2), \ldots, \omega(e_{k-1})\}$ são os *vértices internos de* p e dizemos que um vértice é um *vértice de* p se for extremidade ou um vértice interno de p. Dizemos que p é um *caminho* se os vértices de p são dois a dois distintos. Dados dois vértices v, u dizemos que a distância de v a u é o comprimento do passeio mais curto que inicia em v e termina em u. No grafo da Figura 1.1, o passeio visto anteriormente não é um caminho pois repete o vértice u. Além disso, a distância de w a x é 1, que é o comprimento do caminho formado pela aresta (w, x).

Dizemos que um grafo $\mathbf{T} = (V, E)$ é uma árvore se existir um vértice $r \in V$ (no caso de $V \neq \emptyset$), chamado de *raiz* e denotado por R (**T**), tal que todo vértice $v \in V$ admite um único passeio de r para v. É imediato verificar que este passeio único é também um caminho. A *profundidade* do vértice v é o comprimento deste único caminho e a *profundidade da árvore* **T** é a profundidade de seu vértice mais profundo. Assim, a raiz é o único vértice com profundidade 0. A *árvore trivial* não possui nenhum vértice e nenhuma aresta. Sua profundidade não é definida. A menor árvore não trivial é aquela que não possui nenhuma aresta e um único vértice: a raiz. Dados dois vértices u e v, dizemos que u é *pai* de v e que v é *filho* de u se existir uma aresta de u para v. É imediato verificar que todo vértice, exceto a raiz, é filho de um único vértice-pai, a saber, o início da última aresta do único caminho da raiz até o mesmo. A raiz não é filha de ninguém. Isto

[8]Como no caso das arestas, também com os passeios usaremos termos mais relaxados em relação ao seus inícios e términos.

implica que o número de arestas numa árvore não trivial é um a menos que o número de vértices. Dizemos ainda que o vértice u é um *ancestral* de v se u é um vértice do único passeio da raiz até v e se $u \neq v$. Observe que o pai de v é seu ancestral mais profundo. Cada vértice v que não possui nenhum filho é chamado de *folha*. Equivalentemente, v é uma folha se não for ancestral de nenhum outro vértice. Os vértices que não são folhas são chamados de *vértices internos* da árvore. Note que as folhas de uma árvore com mais de um vértice são os vértices que não são nem raiz nem vértice interno de nenhum caminho.

Ao representar uma árvore, tipicamente, omitiremos as flechas das arestas e estas supostamente começarão em pontos mais altos e terminam em pontos mais baixos da figura. Assim a raiz está sempre no topo[9]. Na Figura 1.2 temos uma árvore de raiz r, profundidade 2, nós internos r, u

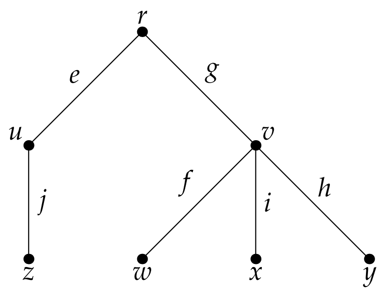


Figura 1.2: Exemplo de árvore

e v, e folhas w, x, y e z. São 6 as arestas e, f, g, h, i e j, enquanto que os vértices são 7. O vértice v é o único pai das folhas w, x, y. Observe-se ainda que o grafo da Figura 1.1 não é uma árvore.

[9]Uma árvore de ponta cabeça!

Capítulo 2

Árvore dos sufixos

Neste capítulo introduziremos uma das estruturas de dados mais interessantes e úteis à elaboração de algoritmos eficientes envolvendo seqüências de qualquer tipo. É a árvore dos sufixos. O principal objetivo do capítulo é a apresentação da mesma, sua caracterização do ponto de vista matemático, bem como a descrição de um algoritmo eficiente que permita a sua construção em tempo linear.

Na Seção 2.1 discutiremos brevemente algumas motivações e características que dão beleza e utilidade à construção linear da árvore de sufixos. Não há aqui uma preocupação estritamente formal e, alguns resultados das seções posteriores serão mencionados, até porque as verdades afirmadas nesta seção não serão usadas nas seções subseqüentes. Esta seção também não tem a pretensão de esgotar as vantagens e aplicações das árvores dos sufixos. Neste sentido, os Capítulos 3 e 5 darão continuidade a parte deste trabalho. Na Seção 2.2 continuamos as definições elementares iniciadas no Capítulo 1, agora com definições já mais específicas. A apresentação aqui feita para a árvore dos sufixos pretende ser bem mais geral que a usualmente vista nas abordagens. O conceito fundamental da seção, as A^+-árvores, tratam-se de autômatos determinísticos especiais onde os estados e as transições têm estrutura de uma árvore e onde os rótulos das transições são palavras quaisquer não vazias, ao invés de letras simples do alfabeto. Na Seção 2.3 por sua vez, veremos a definição de uma árvore de reconhecimento de um conjunto qualquer de palavras, não somente o dos sufixos de uma palavra, como é o caso de uma árvore de sufixos. O Teorema 2.4 dá então uma importante caracterização desta estrutura. Acreditamos que esta caracterização seja nova, frente a abordagem que normalmente temos encontrado na literatura, e acreditamos que seja importante para uma mais clara compreensão das árvores de sufixos. Na Seção 2.4, vemos um conceito-chave à construção eficiente da árvore de reconhecimento. O de cabeça de uma palavra numa árvore. O Lema 2.6 é então fundamental para uma construção algorítmica de sucessivas árvores de reconhecimento feitas no algoritmo 2.2. Na Seção 2.5, vemos a definição propriamente dita da árvore de sufixos e um algoritmo preliminar para a sua construção. Na Seção 2.6, por sua vez, discutimos alguns detalhes mais sobre a implementação dos algoritmos até então vistos e fazemos sua análise. Diante de sua ineficiência assintótica, na Seção 2.7 vemos então outro conceito-chave — a ligação de sufixo — a uma construção eficiente de uma árvore dos sufixos, que será finalmente apresentada à Seção 2.8.

2.1 Motivações

Como o Corolário 2.8 demonstra, a árvore dos sufixos de uma palavra representa não somente o conjunto dos sufixos de uma palavra, mas também o conjunto de *todos* os fatores da mesma. Ademais, uma consulta à árvore é feita em tempo linear ao comprimento do fator, como a Seção 2.6.3 examina.

Para ver o quanto estas características são desejáveis e não imediatamente reconhecidas, vamos primeiro verificar quanta memória num computador poderia ser usada para representar todos os fatores de uma dada palavra w de comprimento n. Observe-se que, para $k = 1, \ldots, n$, temos que w possui $n - k + 1$ fatores de comprimento k, a saber, $w[i + 1 \mathinner{.\,.} i + k]$, para $i = 0, \ldots, n - k$. Sempre desprezando a eventual ocorrência de fatores repetidos, temos então um total de

$$\sum_{k=1}^{n} n - k + 1 = n + (n - 1) + \cdots + 1 = \frac{n(n + 1)}{2}$$

fatores, o que nos dá $O(n^2)$ fatores.

Para armazenar uma palavra u são utilizados normalmente $|u|$ caracteres. Para armazenar os $O(n^2)$ fatores acima vistos, aparentemente seriam necessários S caracteres, onde S é a soma dos comprimentos dos $O(n^2)$

fatores. Ocorre que S é $O(n^3)$. De fato, por um lado,

$$S = \sum_{k=1}^{n} k(n-k+1) \leq \sum_{k=1}^{n} n(n) = n^3,$$

e por outro lado não dá para fazer melhor em termos assintóticos pois[1],

$$\begin{aligned} S &= \sum_{k=1}^{n} k(n-k+1) = 1n + 2(n-1) + \cdots + n1 \\ &\geq \sum_{k=\frac{n}{3}+1}^{\frac{2n}{3}} k(n-k+1) \geq \sum_{k=\frac{n}{3}+1}^{\frac{2n}{3}} \frac{n}{3}(\frac{n}{3}) = (\frac{n}{3})^3 = \frac{n^3}{27}. \end{aligned}$$

É já portanto surpreendente que uma estrutura de dados assim tão rica possa ser armazenada através de um espaço de memória linear, já que o número de vértices de uma árvore de sufixos de comprimento n não excede $2n$ por conseqüência do Lema 2.6. O que prova o Teorema 2.10 é ainda mais surpreendente: que o tempo de processamento necessário à construção da árvore de sufixos seja também linear.

Vale dizer que Donald Knuth, famoso matemático reconhecido como um dos pais da Ciência da Computação, conjecturou [Apo85, KMP77] em 1970 que o problema do mais longo fator comum a duas palavras *não* poderia ser resolvido em tempo linear. Não foi à toa que o primeiro algoritmo linear de construção da árvore de sufixos de uma palavra, desenvolvido por Weiner [Wei73] foi então eleito pelo próprio Knuth como o "algoritmo do ano" [Apo85], talvez até porque o mais longo fator comum a duas seqüências pôde então ser obtido em tempo linear, como mostra a Seção 3.2 do Capítulo 3.

2.2 A^+-Árvores, lugares

Seja um alfabeto A. Definimos um A^+-*grafo* $\mathbf{G} = (V, E, \lambda)$ como sendo um grafo (V, E) e uma rotulação nas arestas $\lambda : E \longrightarrow A^+$. Ademais, exige-se que duas arestas distintas com mesmo início tenham rótulos cujas primeiras letras são também distintas. Isto implica que de cada vértice não partam

[1]Por simplicidade, supomos n divisível por 3. Os outros casos são análogos.

mais que $|A|$ arestas. Dado um passeio $p = e_1e_2 \ldots e_k$, estendemos esta rotulação de forma a definir o *rótulo do passeio p* como sendo a palavra $\lambda(e_1)\lambda(e_2)\cdots\lambda(e_k)$.

Definimos uma A^+-*árvore* como sendo um A^+-*grafo* $\mathbf{T} = (V, E, \lambda)$ em que (V, E) seja também uma árvore. Assim, cada vértice não possui mais que $|A|$ filhos. Como existe exatamente um passeio p da raiz até um vértice qualquer v da árvore, também estendemos a rotulação de forma a definir o *rótulo do vértice v* da A^+-árvore pela palavra $\lambda(p)$. Ademais, o comprimento da palavra $\lambda(p)$ é chamado de *profundidade de palavra* deste vértice v. Uma A^+-árvore é dita *compacta* se cada vértice interno exceto a raiz possui ao menos dois filhos. Na Figura 2.1 temos uma A^+-árvore não compacta com os rótulos das arestas e dos vértices explicitados.

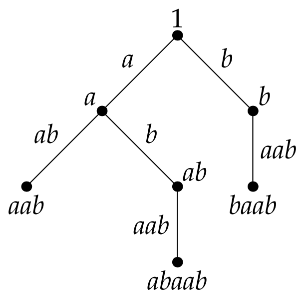


Figura 2.1: Exemplo de A^+-árvore

Seja uma A^+-árvore $\mathbf{T} = (V, E, \lambda)$. Dizemos que uma palavra $w \in A^*$ é uma *palavra de* $\mathbf{T}$ se w for prefixo de um rótulo de um vértice de $\mathbf{T}$. Definimos um *lugar em* $\mathbf{T}$ como sendo um par ordenado (u, x) onde $u \in V$ é um vértice qualquer e $x \in A^*$ é a palavra vazia ou um prefixo próprio não vazio do rótulo de uma aresta que parte de u. Caso x seja vazio, permitimo-nos dizer que o lugar é o vértice u. Caso contrário, dizemos que o lugar está numa aresta, a saber, a única aresta que parte de u e que começa com a primeira letra de x. Também estendemos a rotulação de forma a definir o *rótulo do lugar* (u, x) como sendo a palavra $\lambda((u, x)) = \lambda(u)x$.

Na proposição 2.1 vemos uma propriedade que relaciona a ancestralidade entre dois vértices com os prefixos de seus rótulos e no corolário 2.2 vemos que não há vértices com mesmos rótulos. Na proposição 2.3 vemos que a rotulação λ estabelece uma bijeção entre as palavras de uma A^+-árvore e os lugares na mesma. A proposição 2.3 mostra-nos que a rotulação λ estabelece uma bijeção entre os lugares na A^+-árvore $\mathbf{T}$ e as palavras de $\mathbf{T}$. Assim, dada uma palavra w de $\mathbf{T}$, podemos então definir o *lugar de w em* $\mathbf{T}$ como sendo o lugar $\lambda^{-1}(w) = (u, \lambda(u)^{-1}w)$, onde u é o vértice mais profundo tal que $\lambda(u)$ é prefixo de w.

Proposição 2.1 *Sejam u e v dois vértices de uma A^+-árvore $\mathbf{T} = (V, E, \lambda)$. Então u é um ancestral de v se e só se $\lambda(u)$ for prefixo próprio de $\lambda(v)$.*

Prova. Suponha que e o vértice u seja um ancestral do vértice v. Então o passeio p da raiz até v pode ser decomposto como a concatenação dos passeios q da raiz até u e o passeio não vazio r de u até v. Assim $\lambda(u) = \lambda(q)$ é prefixo próprio de $\lambda(q)\lambda(r) = \lambda(qr) = \lambda(p) = \lambda(v)$.

Suponha agora que $\lambda(u)$ seja um prefixo próprio de $\lambda(v)$. Provaremos que u é ancestral de v por indução em $|\lambda(u)|$. Se $|\lambda(u)| = 0$, temos que u é a raiz. Como v não é a raiz u já que $\lambda(v) \neq 1 = \lambda(u)$, como u é vértice do passeio da raiz a v, temos que u é ancestral de v. Suponhamos agora que $|\lambda(u)| > 0$ e que a hipótese de indução seja válida. Seja w o pai de u. Assim w é o início da última aresta do único caminho da raiz a u e temos que $\lambda(u) = \lambda(w) \cdot \lambda((w, u))$. Ademais, temos que $\lambda(w) \in \mathrm{Pref}(\lambda(u))$ é prefixo próprio de $\lambda(v)$ e w é ancestral de v por hipótese de indução. Assim existem passeios p da raiz a v, q da raiz a w e r de w a v tais que $p = qr$ e $\lambda(v) = \lambda(p) = \lambda(q)\lambda(r) = \lambda(w)\lambda(r)$. Seja a a primeira letra do rótulo da aresta de w a u. Assim, $\lambda(w)a \in \mathrm{Pref}(\lambda(w)\lambda((w, u))) = \mathrm{Pref}(\lambda(u)) \subset \mathrm{Pref}(\lambda(v)) = \mathrm{Pref}(\lambda(w)\lambda(r))$ implica que a seja a primeira letra de $\lambda(r)$, a primeira letra do rótulo da primeira aresta de r. Como existe uma única aresta, a saber (w, u), que parte de w e cujo rótulo começa com a, esta é a primeira aresta de r. Assim u é vértice de r, e portanto de $p = qr$. É imediato que $u \neq v$ já que $\lambda(u)$ é prefixo próprio de $\lambda(v)$. ∎

Corolário 2.2 *Sejam u e v dois vértices de uma A^+-árvore $\mathbf{T} = (V, E, \lambda)$. Então $u = v$ se e só se $\lambda(u) = \lambda(v)$.*

Proposição 2.3 *Seja uma A^+-árvore $\mathbf{T} = (V, E, \lambda)$. A função λ estabelece uma bijeção entre os lugares de $\mathbf{T}$ e as palavras de $\mathbf{T}$. Além disso, se w é uma palavra de $\mathbf{T}$ e u é o vértice mais profundo cujo rótulo seja prefixo de w, então $\lambda^{-1}(w) = (u, \lambda(u)^{-1}w)$.*

Prova. Seja (u, x) um lugar qualquer em $\mathbf{T}$. Suponha que $x = 1$. Assim, $\lambda((u, x)) = \lambda(u)x = \lambda(u)$ é uma palavra de $\mathbf{T}$. Suponha agora que $x \neq 1$. Seja então a aresta $e = (u, v)$ tal que $x \in \mathrm{Pref}(\lambda(e))$. Assim, $\lambda((u, x)) = \lambda(u)x \in \lambda(u)\mathrm{Pref}(\lambda(e)) \subseteq \mathrm{Pref}(\lambda(u)\lambda(e)) = \mathrm{Pref}(\lambda(v))$ é uma palavra de $\mathbf{T}$.

Seja agora w uma palavra de $\mathbf{T}$. Nem que seja a raiz, podemos escolher u o vértice mais profundo cujo rótulo seja prefixo de w. Assim, a palavra $\lambda(u)^{-1}w$ é definida. Primeiramente provaremos que $(u, \lambda(u)^{-1}w)$ é um lugar em $\mathbf{T}$. Suponha primeiro o caso em que $\lambda(u) = w$. Então $\lambda(u)^{-1}w = 1$ e $(u, \lambda(u)^{-1}w)$ é certamente um lugar em $\mathbf{T}$, a saber, o próprio vértice u. Suponha agora o caso em que $\lambda(u)^{-1}w \neq 1$. Como w é uma palavra de $\mathbf{T}$, seja um vértice v' tal que w seja um prefixo de seu rótulo e seja p um passeio da raiz até v'. Como $\mathrm{Pref}(\lambda(u)) \subset \mathrm{Pref}(w) \subseteq \mathrm{Pref}(\lambda(v'))$, temos que u é um ancestral de v' devido à proposição 2.1. Podemos definir v o primeiro vértice após u dentro do passeio p. Devido à escolha de u, temos que $\lambda(v) \notin \mathrm{Pref}(w)$. Como w e $\lambda(v)$ são prefixos de $\lambda(p)$, temos que w é prefixo de $\lambda(v)$. Assim $\lambda(u)^{-1}w$ é prefixo de $\lambda(u)^{-1}\lambda(v)$, que é o rótulo da aresta (u, v). Isto prova que $(u, \lambda(u)^{-1}w)$ é um lugar em $\mathbf{T}$, que está na aresta (u, v) neste caso. Por fim, $\lambda((u, \lambda(u)^{-1}w)) = \lambda(u)\lambda(u)^{-1}w = w$. ∎

2.3 A^+-árvore de reconhecimento

Seja W um conjunto de palavras. Dizemos que uma A^+-árvore $\mathbf{T} = (V, E, \lambda)$ *reconhece* W se existir um subconjunto de vértices $X \subseteq V$ tal que $\lambda(X) = \{\lambda(x) \mid x \in X\} = W$. Como não há dois vértices distintos com o mesmo rótulo devido ao corolário 2.2, o conjunto X é unicamente determinado em função de W e vice-versa. Dizemos ainda que $\mathbf{T} = (V, E, \lambda)$ *reconhece W em X* e chamamos cada vértice de X de *vértice final*[2]. Tipicamente, os vértices finais a serem considerados serão as folhas. Nalguns contextos contudo, outros vértices também serão considerados finais. As folhas da A^+-árvore da Figura 2.2 são vértices finais e por isto estão destacadas. Nem sempre quereremos destacar os vértices finais, mas quando o fizermos será como nesta figura. Devido à unicidade dos rótulos dos vértices garantida pelo corolário 2.2, com freqüência identificaremos os vértices com seus rótulos, como é o caso nesta figura. Observemos que o conjunto W das palavras reconhecidas pelo conjunto X das folhas da A^+-árvore da Figura 2.2 é o

[2] Naturalmente que este atributo supõe fixado um particular conjunto W que se deseja reconhecer.

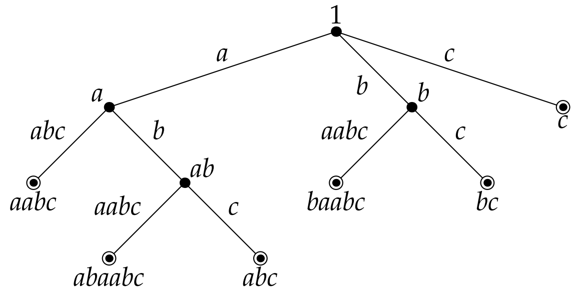


Figura 2.2: A^+-árvore de reconhecimento de $\mathrm{Suf}(abaabc) \setminus \{1\}$

conjunto

$$\{abaabc, baabc, aabc, abc, bc, c\},$$

ou seja, o conjunto $\mathrm{Suf}(abaabc) \setminus \{1\}$ dos sufixos não vazios da palavra *abaabc*. Observemos ainda que a árvore da figura é compacta e que o fato da última letra c ocorrer uma única vez implica que não haja dois de tais sufixos tais que um deles seja prefixo do outro.

Dado um conjunto de palavras W, dizemos que W é *livre de prefixos* se não existirem duas palavras distintas em W tais que uma delas seja prefixo da outra. É imediato verificar que, dada uma palavra w, o conjunto dos sufixos não vazios de w é livre de prefixos se e só se a última letra de w não ocorre noutra posição, como é o caso da palavra *abaabc*. Dizemos que uma palavra x é *prefixo de* W se x for prefixo de alguma palavra de W. Dizemos ainda que x é uma *bifurcação de* W se x for prefixo de W e se existirem letras distintas $a, b \in A$ tais que xa e xb também sejam prefixos de W. Observe que isto implica que W tenha ao menos duas palavras não vazias já que xa e xb não são prefixos da palavra vazia e não podem ser prefixos de uma mesma palavra. O *conjunto das bifurcações de* W é denotado por $\mathrm{Bifurc}(W)$. Definimos o *conjunto unitário de* W $\mathrm{Um}(W)$ como sendo $\emptyset$ se $W = \emptyset$ ou se $W = \{1\}$ e $\mathrm{Um}(W) = \{1\}$ caso contrário. Dizemos que uma palavra x é *prefixo próprio de* W se x for prefixo próprio de alguma palavra de W. Como uma bifurcação de W é prefixo próprio de W, $\mathrm{Bifurc}(W)$ é disjunto de W se W for livre de prefixos. Observe que o conjunto $\mathrm{Suf}(abaabc) \setminus \{1\}$ reconhecido pela A^+-árvore da Figura 2.2 é livre de prefixos e que cada rótulo de cada vértice interno é uma bifurcação deste conjunto. Podemos generalizar esta construção para um conjunto qualquer de palavras W e definimos a

A^+-árvore de reconhecimento de W como sendo a A^+-árvore $\mathbf{T} = (V, E, \lambda)$ onde

$$
\begin{aligned}
&V = W \cup \mathrm{Bifurc}(W) \cup \mathrm{Um}(W)\\
&E = \{(u, v) \in V \times V \mid u = \max_{|x|}\{x \in V \mid x \text{ é prefixo próprio de } v\}\}\\
&\begin{array}{rccc} \lambda: & E & \longrightarrow & A^+\\ & (u,v) & \longrightarrow & u^{-1}v. \end{array}
\end{aligned}
$$

De fato, o Teorema 2.4 prova que $\mathbf{T} = (V, E, \lambda)$ assim definida é de fato uma A^+-árvore, que reconhece W e que as palavras de $\mathbf{T}$ são exatamente os prefixos de W. A A^+-árvore da Figura 2.2 é a A^+-árvore de reconhecimento do conjunto $\mathrm{Suf}(abaabc) \setminus \{1\}$.

Teorema 2.4 *Seja W um conjunto qualquer de palavras, seja $\mathbf{T} = (V, E, \lambda)$ sua A^+-árvore de reconhecimento e seja $F \subseteq W$ o conjunto das palavras de W que não são prefixos próprios de W. Então:*

1. *$\mathbf{T}$ é árvore trivial se e só se $W = \emptyset$;*
2. *$\mathbf{T}$ é uma A^+-árvore (de raiz 1 se não for trivial);*
3. *$\lambda(v) = v$ para todo vértice v;*
4. *$\mathbf{T}$ reconhece W (com vértices finais W);*
5. *o conjunto das palavras de $\mathbf{T}$ é o conjunto dos prefixos de W;*
6. *as folhas de $\mathbf{T}$ são F e os nós internos são $\mathrm{Bifurc}(W) \cup \mathrm{Um}(W) \cup (W \setminus F)$;*
7. *os nós internos com pelo menos dois filhos são $\mathrm{Bifurc}(W)$;*

Prova. A partir da definição, temos claramente que $V = \emptyset$ se e só se $W = \emptyset$, o que prova o item 1.

Vamos provar o item 2. Caso $W = \emptyset$, temos que $\mathbf{T}$ é A^+-árvore trivial. Suporemos que $W \neq \emptyset$ a partir de agora. Seja $v \in V$. Se $v = 1$, como não há prefixo próprio de v temos que v não é término de nenhuma aresta. Se $v \neq 1$, temos que $1 \in V$ é um prefixo próprio de v e que v é término de uma única aresta: daquela que parte de u, seu prefixo próprio mais comprido que está em V. Por indução no comprimento de v podemos provar que existe um único caminho de 1 a v em $\mathbf{T}$: aquele caminho de 1 a u mais a aresta (u, v). Isto prova que $\mathbf{T}$ é árvore com raiz 1. As arestas (u, v) têm rótulos não vazios $u^{-1}v$ já que u é prefixo próprio de v. Sejam duas arestas distintas (u, v) e (u, v'). Seja x o mais comprido prefixo comum de v e de v'. Suponha por absurdo que $x = v'$. Da escolha da aresta (u, v'), temos que $\mathrm{Pref}(u) \subset \mathrm{Pref}(v')$. Da escolha de x, temos que $\mathrm{Pref}(v') = \mathrm{Pref}(x) \subseteq \mathrm{Pref}(v)$. Como as arestas (u, v) e (u, v') são distintas, temos que $\mathrm{Pref}(v) \neq \mathrm{Pref}(v')$ e, portanto, $\mathrm{Pref}(u) \subset \mathrm{Pref}(v') \subset \mathrm{Pref}(v)$ e u não é a palavra de V mais comprida que é prefixo próprio de v, o que contradiz com a escolha da aresta (u, v). Donde, $x \neq v'$. Analogamente, $x \neq v$. Da escolha de x, temos que a primeira letra de $x^{-1}v$, chamemo-la a, é distinta da primeira letra de $x^{-1}v'$, chamemo-la b. Como $V \subseteq \mathrm{Pref}(W)$, temos que $xa \in \mathrm{Pref}(x\,x^{-1}v) = \mathrm{Pref}(v) \subseteq \mathrm{Pref}(W)$ e que $xb \in \mathrm{Pref}(x\,x^{-1}v') = \mathrm{Pref}(v') \subseteq \mathrm{Pref}(W)$. Assim $x \in \mathrm{Bifurc}(W) \subseteq V$. Como u é um prefixo comum de v e de v', da escolha de x temos que $\mathrm{Pref}(u) \subseteq \mathrm{Pref}(x) \subset \mathrm{Pref}(v)$. Do fato de u ser o prefixo mais comprido de v que está em V e $x \in V$, temos que $x = u$, que a é a primeira letra de $x^{-1}v = u^{-1}v = \lambda((u, v))$, e que b é a primeira letra de $x^{-1}v' = u^{-1}v' = \lambda((u, v'))$ e que $a \neq b$. Isto completa a prova de que $\mathbf{T}$ é uma A^+-árvore de raiz 1.

Vamos provar os ítens 3 e 4. Seja $v_0, v_1, v_2, \ldots, v_k$, com $1 = v_0$ e $v_k = v$, a seqüência de vértices percorridos pelo passeio p da raiz até um vértice qualquer v. Então

$$\begin{aligned}
\lambda(v) &= \lambda(p) \\
&= \lambda((v_0, v_1))\,\lambda((v_1, v_2))\,\lambda((v_2, v_3)) \cdots \lambda((v_{k_1}, v_k)) \\
&= (v_0^{-1}v_1)(v_1^{-1}v_2)(v_2^{-1}v_3) \cdots (v_{k-1}^{-1}v_k) \\
&= v_0^{-1}v_k = 1^{-1}v \\
&= v.
\end{aligned}$$

Assim, $\lambda(v) = v$ e $W \subseteq V$ é reconhecida por $\mathbf{T}$ em $X = W$.

Vamos provar o item 5. Seja w uma palavra de $\mathbf{T}$ e seja $v \in V$ tal que $w \in \mathrm{Pref}(\lambda(v))$. Assim, $w \in \mathrm{Pref}(\lambda(v)) = \mathrm{Pref}(v) \subseteq \mathrm{Pref}(W)$ já que toda palavra em V é prefixo de W. Seja agora u um prefixo de W e seja $w \in W$ tal que $u \in \mathrm{Pref}(w)$. Como $w \in W \subseteq V$ e $u \in \mathrm{Pref}(w) = \mathrm{Pref}(\lambda(w))$, temos que u é palavra de $\mathbf{T}$.

Vamos provar o item 6. Suponha que $u \in V$ seja uma folha. Assim u não é ancestral de nenhum vértice em W e, devido à proposição 2.1, $\lambda(u) = u$ não é prefixo próprio de $\lambda(w) = w$ para nenhum $w \in W$. Assim $u \in F$. Suponha agora que $u \in V$ seja um vértice interno. Assim u é pai de algum vértice-filho $v \in V$ e existe a aresta (u, v). Assim, u é prefixo próprio de $v \in V \subseteq \mathrm{Pref}(W)$, e $u \notin F$. Isto prova que as folhas de $\mathbf{T}$ são F. Como

$F \cap \text{Bifurc}(W) = \emptyset$ já que toda bifurcação de W é prefixo próprio de W, como $\text{Um}(W) \cap F = \emptyset$ pois 1 é prefixo próprio de W sempre que W possui uma palavra não vazia, temos que os vértices internos são

$$\begin{aligned} V \setminus F &= (\text{Bifurc}(W) \cup \text{Um}(W) \cup W) \setminus F \\ &= ((\text{Bifurc}(W) \cup \text{Um}(W)) \setminus F) \cup (W \setminus F) \\ &= (\text{Bifurc}(W) \cup \text{Um}(W)) \cup (W \setminus F). \end{aligned}$$

Vamos provar o item 7. Seja $u \in \text{Bifurc}(W)$ um vértice interno. Como u é uma bifurcação de W, sejam a e b letras distintas tais que ua e ub sejam prefixos de W. Assim ua é palavra de $\mathbf{T}$. Suponha o caso em que $ua \in V$. Neste caso, (u, ua) é certamente uma aresta de E cujo rótulo começa com a. Suponha agora o caso em que $ua \notin V$. Neste caso, usando a proposição 2.3, o lugar de ua em $\mathbf{T}$ é o lugar (u, a), que está na aresta que parte de u e cujo rótulo começa com a. Em qualquer caso, existe uma aresta que parte de u e cujo rótulo começa com a. Analogamente, existe uma aresta que parte de u e cujo rótulo começa com b. Assim u tem pelo menos dois filhos. Seja x um vértice interno com pelo menos dois filhos, y e z. Sejam a a primeira letra do rótulo da aresta (x, y) e b a primeira letra do rótulo da aresta (x, z). Como $\mathbf{T}$ é A^+-árvore temos que $a \neq b$. Por definição, (x, a) e (x, b) são dois lugares em $\mathbf{T}$. Pela proposição2.3, temos que $\lambda((x, a)) = \lambda(x)a = xa$ e $\lambda((x, b)) = xb$ são duas palavras de $\mathbf{T}$, e portanto são dois prefixos de W. Assim, x é uma bifurcação de W, completando a prova. ∎

Na Figura 2.3 temos a A^+-árvore de reconhecimento dos sufixos não vazios de *abaab*, que não é livre de prefixos, já que *ab* é prefixo de *abaab*.

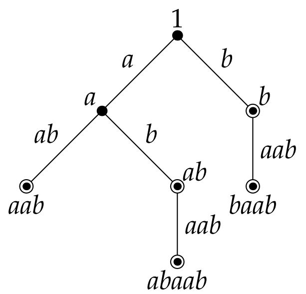


Figura 2.3: A^+-árvore de reconhecimento de $\text{Suf}(abaab) \setminus \{1\}$

Observe que a árvore não é compacta, que as folhas são $\{abaab, baab, aab\}$, que as bifurcações são $\{1, a\}$ e têm dois filhos e que $\{ab, b\} = \mathrm{Suf}(abaab) \setminus \{1\} \setminus \{abaab, baab, aab\}$ são os vértices com um único filho.

No caso em que $W \neq \{1\}$ é livre de prefixos temos que $F = W$ e segue de imediato o corolário 2.5.

Corolário 2.5 *Se além das hipóteses do teorema anterior tivermos que W é livre de prefixos, temos então que as folhas são W e os vértices internos são* $\mathrm{Bifurc}(W) \cup \mathrm{Um}(W)$ *e* $\mathbf{T}$ *é uma A^+-árvore compacta.*

2.4 Cabeças e Caudas

Mesmo que uma palavra w não seja uma palavra da A^+-árvore não trivial $\mathbf{T}$, existe sempre um prefixo mais comprido de w que o seja (no pior caso, a palavra vazia). Para toda palavra $w \in A^*$ podemos fatorá-la em $w = ht$ tal que h seja este prefixo mais comprido. Assim, h é definido como a *cabeça de w em* $\mathbf{T}$ e t é definida como sendo a *cauda de w em* $\mathbf{T}$. No Algoritmo 2.1 vemos a função S que aplicada sobre (w, raiz de $\mathbf{T}$, $\mathbf{T}$) devolve a cabeça de w em $\mathbf{T}$, bem como seu lugar em $\mathbf{T}$ e a aresta associada. Se w for já uma

---

Algoritmo 2.1 Função S

S $(w, r, \mathbf{T})$

▹ w: palavra a ser soletrada a partir do
▹ r: vértice de origem, tipicamente, a raiz
▹ $\mathbf{T}$: a A^+-árvore em questão
▹ devolve a cabeça de $\lambda(r)w$ em $\mathbf{T}$, seu lugar e aresta associada
$u \leftarrow r$
$s \leftarrow w$
enquanto $|s| > 0$ **e** $\exists e = (u, v) \mid \lambda(e) = \lambda(u)^{-1}\lambda(v) \in \mathrm{Pref}(s)$ **faça**
    $s \leftarrow \lambda(e)^{-1}s$
    $u \leftarrow v$
se $|s| > 0$ **e** $\exists e = (u, v) \mid s[1] \in \mathrm{Pref}(\lambda(e))$ **então**
    $x \leftarrow$ o mais comprido prefixo de $\lambda(e)$ e de s
    ▹ $\lambda(u)x$ é prefixo próprio de $\lambda(v)$
senão $x \leftarrow 1$
    $v \leftarrow u$
devolva $(\lambda(u)x, u, x, v)$

---

palavra de $\mathbf{T}$, é claro que a cabeça de w em $\mathbf{T}$ será a própria palavra w e S $(w, \text{raiz de } \mathbf{T}, \mathbf{T})$ terá o feito de "soletrar" as letras de w de forma a obter o lugar de w em $\mathbf{T}$.

O Lema 2.6 dá-nos uma importante comparação entre os conjuntos de vértices das A^+-árvores de reconhecimento $\mathbf{T}$ e $\mathbf{T}'$ de conjuntos W e $W' = W \cup \{w\}$, respectivamente. Os vértices w e h (a cabeça de w em $\mathbf{T}$, possivelmente igual a w) são os vértices que são acrescentados (se ainda não presentes) a $\mathbf{T}$ para a obtenção de $\mathbf{T}'$.

Lema 2.6 *Sejam W um conjunto de palavras não vazio, $\mathbf{T} = (V, E, \lambda)$ sua A^+-árvore de reconhecimento, $w \notin W$ uma palavra, $\mathbf{T}' = (V', E', \lambda')$ a A^+-árvore de reconhecimento de $W' = W \cup \{w\}$, h a cabeça de w em $\mathbf{T}$, e $F \subseteq W$ o conjunto das palavras de W que não são prefixos próprios de W. Então temos que*

$$\begin{aligned} V' &= V \cup \{h, w\} \\ h = w &\iff w \in \mathrm{Pref}(W) \\ h \in F &\iff w \in FA^* \\ h \in \mathrm{Bifurc}(W') \setminus \{w\} &\iff w \notin \mathrm{Pref}(W) \cup FA^*. \end{aligned}$$

Prova. Seja t a cauda de w em $\mathbf{T}$. Como h é o mais longo prefixo de w que está em $\mathbf{T}$, usando o item 5 do Teorema 2.4, temos que h é o mais longo prefixo de w que é prefixo de W. Em particular, $h = w$ se e só se $w \in \mathrm{Pref}(W)$.

Vamos provar que $h \in F \iff w \in FA^*$. Suponha que $h \in F$. Como $h \in \mathrm{Pref}(w)$, segue da definição que $w \in hA^* \subseteq FA^*$. Suponha que $w \in FA^*$. Seja $x \in F$ tal que x é prefixo de w. Como $x \in F \subseteq W \subseteq \mathrm{Pref}(W)$, como h é o mais longo prefixo de w que é prefixo de W, segue que $x \in \mathrm{Pref}(h) \subseteq \mathrm{Pref}(W)$. Como $x \in F$ não é prefixo próprio de W, x não é prefixo próprio de h. Portanto, $h = x \in F$.

Vamos provar que $h \in \mathrm{Bifurc}(W') \setminus \{w\}$ se e só se $w \notin \mathrm{Pref}(W) \cup FA^*$. Suponha o caso em que $w \notin \mathrm{Pref}(W) \cup FA^*$. Como $w \notin \mathrm{Pref}(W)$, temos que $h \neq w$ e que $t = h^{-1}w \neq 1$. Seja a a primeira letra de t. Como h é o mais longo prefixo de w que é prefixo de W e $ha \in \mathrm{Pref}(ht) = \mathrm{Pref}(w)$, temos que $ha \notin \mathrm{Pref}(W)$. Como $ht = w \notin FA^*$, temos que $h \notin F$. Assim, h é prefixo próprio de W e existe letra b tal que $hb \in \mathrm{Pref}(W) \subseteq \mathrm{Pref}(W')$. Naturalmente que $a \neq b$ já que $ha \notin \mathrm{Pref}(W)$. Como $ha \in \mathrm{Pref}(w) \subseteq \mathrm{Pref}(W')$, temos por definição que $h \in \mathrm{Bifurc}(W')$. Suponha o caso em que $h \in \mathrm{Bifurc}(W') \setminus \{w\}$. Como já vimos, $h \neq w$ implica que $w \notin \mathrm{Pref}(W)$. Sejam a e b duas letras distintas tais que ha e hb sejam prefixos de $W' = W \cup \{w\}$. Como $ha, hb \in \mathrm{Pref}(w)$ implicaria que $a = b$, temos que existe letra $c \in \{a, b\}$ tal que $hc \in \mathrm{Pref}(W)$. Assim, h é prefixo próprio de W, $h \notin F$ e, como já vimos, $w \notin FA^*$.

Vamos provar que $\mathrm{Bifurc}(W') \setminus \mathrm{Bifurc}(W) \subseteq \{h\}$. Seja $x \in \mathrm{Bifurc}(W') \setminus \mathrm{Bifurc}(W) = \mathrm{Bifurc}(W \cup \{w\}) \setminus \mathrm{Bifurc}(W)$. Assim existem letras a, b com $a \neq b$ tais que $x, xa, \in \mathrm{Pref}(w)$ e $x, xb \in \mathrm{Pref}(W)$. Como $a \neq b$, como $x, xb \in \mathrm{Pref}(W)$, segue que $xa \notin \mathrm{Pref}(W)$ pois, do contrário, x seria uma bifurcação de W, contradizendo com sua escolha. Como $xa \in \mathrm{Pref}(w)$, segue que x é o mais longo prefixo de w que é prefixo de W. Portanto, $x = h$.

Vamos por fim provar que $V' = V \cup \{h, w\}$. Primeiro provaremos que $V' \supseteq V \cup \{h, w\}$. Naturalmente que $w \in W' \subseteq V'$. Caso $w \in \mathrm{Pref}(W)$, temos que $h = w \in V'$. Caso $w \in FA^*$, temos que $h \in F \subseteq W \subseteq W' \subseteq V'$. Caso $w \notin \mathrm{Pref}(W) \cup FA^*$, temos que $h \in \mathrm{Bifurc}(W') \setminus \{w\} \subseteq V'$. Em suma, $h \in V'$ em qualquer dos três casos acima. Ora, as próprias definições implicam que $W \subseteq W'$, que $\mathrm{Bifurc}(W) \subseteq \mathrm{Bifurc}(W')$ e que $\mathrm{Um}(W) \subseteq \mathrm{Um}(W')$. Portanto, $V = W \cup \mathrm{Bifurc}(W) \cup \mathrm{Um}(W) \subseteq W' \cup \mathrm{Bifurc}(W') \cup \mathrm{Um}(W') = V'$. Agora provaremos que $V' \subseteq V \cup \{h, w\}$. Como $W \neq \emptyset$, temos que $\mathbf{T}$ é árvore não trivial devido ao item 1 do Teorema 2.4 e, portanto, que $1 \in V$. Como $V = W \cup \mathrm{Bifurc}(W) \cup \{1\}$, segue finalmente que

$$
\begin{aligned}
V' &= V \cup (V' \setminus V) \\
&= V \cup ((W' \cup \mathrm{Bifurc}(W') \cup \mathrm{Um}(W')) \setminus V) \\
&= V \cup (W' \setminus V) \cup (\mathrm{Bifurc}(W') \setminus V) \cup (\mathrm{Um}(W') \setminus V) \\
&\subseteq V \cup (W' \setminus W) \cup (\mathrm{Bifurc}(W') \setminus \mathrm{Bifurc}(W)) \cup (\mathrm{Um}(W') \setminus \{1\}) \\
&\subseteq V \cup \{w\} \cup \{h\} \cup \emptyset.
\end{aligned}
$$

∎

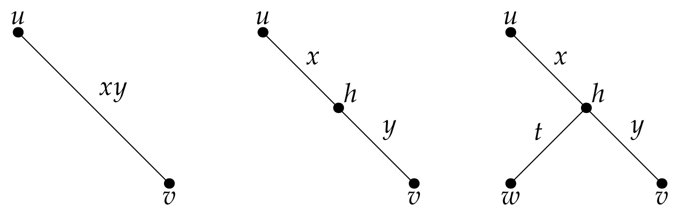
No caso em que $W = \emptyset$, sua A^+-árvore de reconhecimento é trivial. No caso em que $W = \{1\}$, sua A^+-árvore de reconhecimento é formada do único vértice $\{1\}$ e não possui nenhuma aresta. No caso em que $W = \{w\}$, com $w \neq 1$, sua A^+-árvore de reconhecimento possui vértices $\{1, w\}$ e a única aresta $(1, w)$, cujo rótulo é w. Usando o Lema 2.6, é imediato verificar que o Algoritmo 2.2 acrescenta uma palavra w ao conjunto das palavras reconhecidas por uma A^+-árvore de reconhecimento $\mathbf{T}$. O algoritmo A - P descreve que operações em $\mathbf{T}$ são realizadas. Se w é palavra de $\mathbf{T}$, sua cabeça h é igual a w e apenas a inserção do vértice h é feita, caso já não ocorra de h ser um vértice ($|x| = 0$). Se w não é palavra de $\mathbf{T}$,

Algoritmo 2.2 Adição de w a uma A^+-árvore de reconhecimento **T**

```
A        P        (T, w)
▹ T: A⁺-árvore de reconhecimento de um conjunto de palavras W
▹ w: palavra qualquer
▹ devolve A⁺-árvore de reconhecimento de W ∪ {w}
se T é árvore trivial então
        se |w| = 0 então
                acrescente uma raiz 1
        senão   acrescente uma raiz 1 e novo vértice w
                acrescente aresta de 1 para w de rótulo w
senão   (h, u, x, v) ← S        (w, 1, T)        ▹ h=ux
        t ← h⁻¹w
        se |x| > 0 então
                ▹ (u, v) é a aresta associada ao lugar (u, x)
                ▹ x = u⁻¹h ∈ Pref(u⁻¹v)
                acrescente novo vértice h
                remova a aresta (u, v)
                acrescente aresta de u para h de rótulo x
                acrescente aresta de h para v de rótulo h⁻¹v
        se |t| > 0 então
                acrescente aresta de h para nova folha w de rótulo t
devolva T
```

além da possível inserção do vértice h, é feita a inserção do vértice w. As arestas correspondentes também devem ser inseridas. Observe que após as execuções das linhas 9 e 10, h e t são a cabeça e a cauda de w em **T** e (h, x) é o lugar de h em **T**. Ademais, (u, v) é a aresta associada ao lugar (h, x) se ele não for um vértice.

Na Figura 2.4 temos uma ilustração de alguns trechos do algoritmo A P . Seja $y = h^{-1}v$. Na Figura 2.4(a), temos a parte relevante da A^+-árvore **T** antes a execução da linha 14. Na Figura 2.4(b), temos a parte relevante da A^+-árvore **T** após a execução da linha 17, ou seja, após a inserção de um novo vértice h, remoção de uma aresta (u, v) e inserção de duas novas arestas (u, h) e (h, v). Na Figura 2.4(c), temos a parte relevante desta mesma A^+-árvore **T** após a execução da linha 19, ou seja, após a inserção de um novo vértice w e inserção de nova aresta (h, w).



(a) Aresta (u, v) de rótulo xy

(b) Quebra de (u, v) em (u, h) mais (h, v)

(c) Aresta (h, w) de rótulo t

Figura 2.4: Quebrando arestas

2.5 Árvore dos sufixos

Dada uma palavra w, definimos a *árvore dos sufixos de* w como sendo a A^+-árvore de reconhecimento de $\mathrm{Suf}(w)$. Se $w \neq 1$, observe que árvore dos sufixos de w é a mesma A^+-árvore de reconhecimento de $\mathrm{Suf}(w) \setminus \{1\}$. É-nos interessante fazer ainda outra generalização, a ser melhor explorada na Seção 2.8. Dado um conjunto de palavras W, definimos a *árvore dos sufixos de* W como sendo a A^+-árvore de reconhecimento de $\mathrm{Suf}(W)$. Observe que se $W = \{w\}$ contém uma única palavra w, esta será a árvore dos sufixos de w. Em muitos trabalhos a árvore de sufixos de W é chamada de *generalizada*. Vale observar que, por motivos históricos, normalmente as definições encontradas na literatura impõem que a palavra w seja não vazia, que sua última letra seja distinta das demais, e que apenas os sufixos não vazios sejam considerados. Isto garante que $\mathrm{Suf}(w) \setminus \{1\}$ seja livre de prefixos, como vimos. Consideramos que estas restrições sejam artificiais damos uma apresentação mais geral.

Dado um conjunto de palavras W, dizemos que W é *fechado por prefixos* se todo prefixo de W está em W. Equivale a dizer que $\mathrm{Pref}(W) = W$. De maneira análoga, W é dito *fechado por sufixos* se $\mathrm{Suf}(W) = W$ e *fechado por fatores* se $\mathrm{Fat}(W) = W$. Se uma palavra w é uma palavra de uma A^+-árvore, segue da definição que um prefixo de w é também uma palavra da A^+-árvore. Assim, o conjunto das palavras de uma A^+-árvore é fechado por prefixos. Afirmativa análoga não vale para sufixos. Contudo, se W for fechado por sufixos, a proposição 2.7 mostra que o conjunto das palavras de sua A^+-árvore de reconhecimento é $\mathrm{Fat}(W)$, que naturalmente é fechado por prefixos, sufixos e fatores. Ela mostra, em particular, que o conjunto de vértices de uma árvore dos sufixos é fechado por sufixos. O Corolário 2.8 por sua vez, mostra-nos um dos principais interesses de uma árvore dos sufixos de uma palavra: não tanto o de reconhecer o conjunto de seus sufixos, mas principalmente o de ser uma maneira compacta de representar todos os seus fatores.

Proposição 2.7 *Seja W um conjunto de palavras fechado por sufixos e seja $\mathbf{T} = (V, E, \lambda)$ sua A^+-árvore de reconhecimento. Então o conjunto das palavras de $\mathbf{T}$ é* $\mathrm{Fat}(W)$. *Ademais, V é fechado por sufixos.*

Prova. Como W é fechada por sufixos, temos que $W = \mathrm{Suf}(W)$. Como todo fator é um prefixo de um sufixo, do item 5 do Teorema 2.4 temos que o conjunto das palavras de $\mathbf{T}$ é $\mathrm{Pref}(W) = \mathrm{Pref}(\mathrm{Suf}(W)) = \mathrm{Fat}(W)$, que naturalmente é fechado por prefixos, sufixos e fatores.

Seja $x = yz \in \mathrm{Bifurc}(W)$. Sejam $a, b \in A$ tais que xa e xb sejam prefixos de W. Sejam pois $u, v \in A^*$ tais que $xau, xbv \in W$. Assim, $zau \in \mathrm{Suf}(yzau) = \mathrm{Suf}(xau) \subseteq \mathrm{Suf}(W) \subseteq W$. Analogamente, $zbv \in W$, o que implica que $z \in \mathrm{Bifurc}(W)$. Segue de imediato que $V = W \cup \mathrm{Bifurc}(W) \cup \mathrm{Um}(W)$ é fechado por sufixos. ∎

Para $W = \{w\}$, o corolário seguinte mostra que $\mathrm{Fat}(w)$ é o conjunto das palavras da árvore dos sufixos de w.

Corolário 2.8 *O conjunto das palavras da árvore dos sufixos de um conjunto de palavras W é* Fat(W)*, o conjunto dos fatores das palavras de W.*

Prova. Segue de imediato do fato que a árvore dos sufixos de W é a A^+-árvore de reconhecimento de Suf(W), da Proposição 2.7, e do fato que Fat(Suf(W)) = Fat(W). ∎

A seguir, no Algoritmo 2.3, vemos a descrição da função Á D - S que obtém a árvore dos sufixos de w. Para tanto, ele faz uso da tranformação numa A^+-árvore de reconhecimento **T**, A P - , descrita no Algoritmo 2.2. A cada iteração da linha 5, acrescenta-se um sufixo de w ao conjunto das palavras reconhecidas pela A^+-árvore de reconhecimento **T**, do mais comprido para o menos comprido.

Algoritmo 2.3 Construção da árvore dos sufixos

Á D S (w)
1 ▹ w: palavra não vazia cuja árvore dos sufixos será calculada
2 $n \leftarrow |w|$
3 **T** ← árvore trivial
4 **para** i **de** 1 **até** $n + 1$ **faça**
5 **T** ← A P (**T**, $w[i\,..\,n]$)
6 **devolva T**

Na Figura 2.5 vemos as sucessivas A^+-árvores de reconhecimento **T** obtidas durante a execução do Algoritmo 2.3 para a palavra $w = abaabc$. A árvore $\mathbf{T}_0$ é a instância de **T** após a execução da linha 3 do Algoritmo 2.3 enquanto que as árvores $\mathbf{T}_i$, para $i = 1, 2, \ldots, 7$, correspondem às instâncias de **T** após a inserção do sufixo $w[i\,..\,6]$ ao conjunto das palavras reconhecidas por **T** feito à linha 5. A A^+-árvore de reconhecimento $\mathbf{T}_7$ é a árvore dos sufixos de *abaabc*.

2.6 Implementação e Análise

Nos Algoritmos 2.3 e 2.2 vimos as funções Á D S e A - P que descrevem um método utilizado para a construção de uma árvore dos sufixos de uma dada palavra w. É nosso objetivo nesta seção descrever alguns detalhes de implementação dos referidos métodos

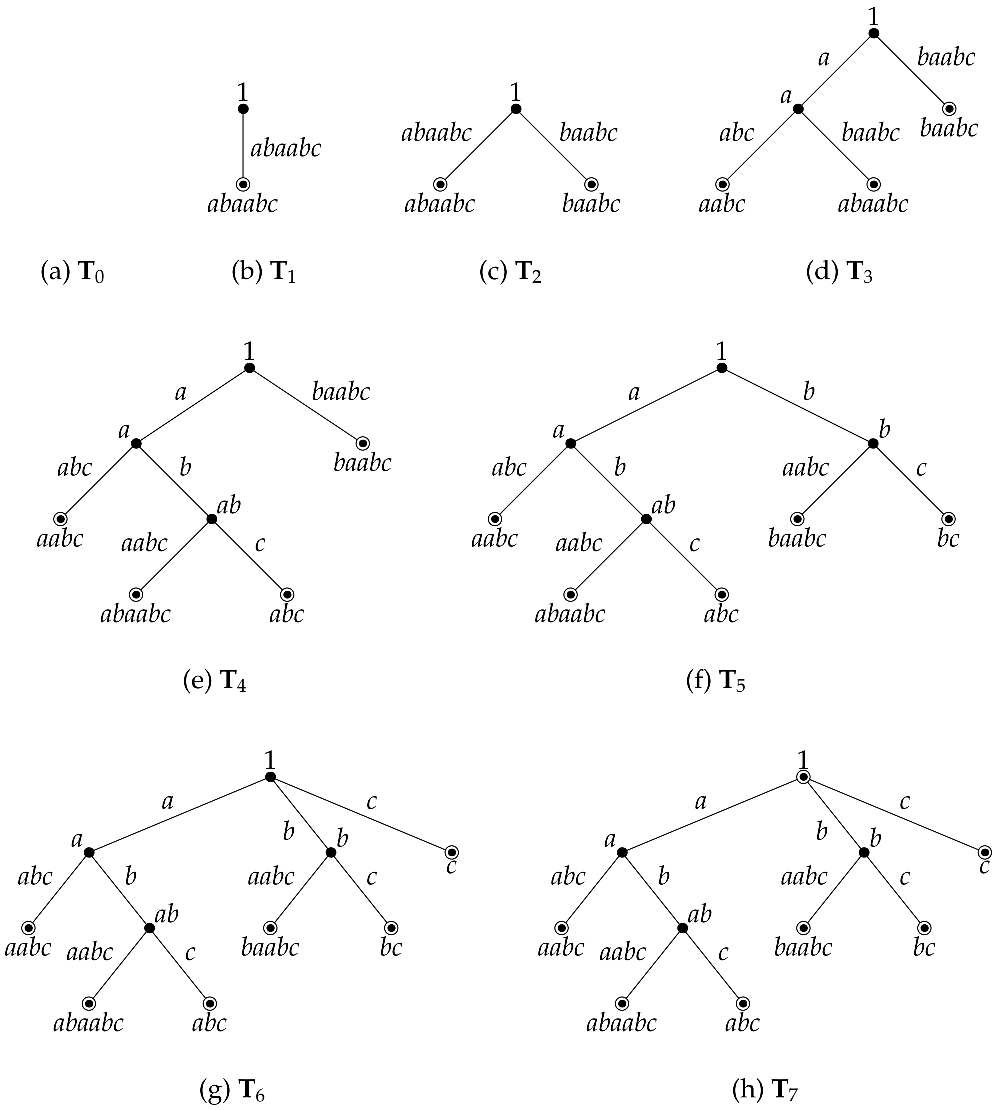


Figura 2.5: A^+-árvores de Á D S (*abaabc*)

de forma que possa então ser feita uma análise de eficiência dos mesmos. Proporemos também melhorias de forma que a construção da árvore dos sufixos de w possa ser feita em tempo linear ao comprimento de w.

Como veremos, duas técnicas de implementação serão fundamentais à elaboração de um algoritmo eficiente neste caso: uma maneira compacta de representação de fatores de w a ser vista na Seção 2.6.1 e; uma maneira de calcular a cabeça de w em **T** em tempo amortizado constante a ser vista na Seção 2.7.

A menos que dito de outra forma, nesta seção e na Seção 2.7, w refere-se a uma palavra não vazia cuja árvore dos sufixos desejamos eficientemente construir. Definimos seu comprimento $n = |w|$.

2.6.1 Representação compacta dos fatores de w

Cada palavra w que é inserida pela função A P , bem como cada rótulo (de vértice, aresta ou lugar), é fator da palavra w (da função Á D S) cuja árvore dos sufixos está sendo computada. Para representar cada uma destas palavras não seria adequado fazê-lo através de um novo vetor de caracteres contendo uma cópia do fator de w em questão. Só para representar os sufixos de w, seria necessário uma cópia de $0 + 1 + 2 + \cdots + n = n(n + 1)/2$ caracteres, o que impossibilitaria a construção da árvore dos sufixos em tempo linear ($O(n)$). Cada fator x de w, inclua-se portanto cada rótulo, será representado através da posição de *uma* ocorrência o de x em w e seu comprimento $l = |x|$. Assim, temos que $x =$ P (o, l), onde a função P é definida como segue:

P (o, l)
▹ o: ocorrência do fator de w
▹ l: comprimento do fator
devolva $w[o \,.\,.\, o + l - 1]$

Observe-se que através desta representação compacta de um fator de w, precisamos apenas de dois inteiros, ocorrência e comprimento, para representar qualquer fator de w.

Sejam u e v fatores de w que ocorrem nas posições o_u e o_v, repectivamente, com comprimentos l_u e l_v, respectivamente. Observe que se $u \in \mathrm{Pref}(v)$, temos pois que $u =$ P (o_v, l_u) e que $u^{-1}v =$ P $(o_v + l_u, l_v - l_u)$. Por outro lado, se $u \in \mathrm{Suf}(v)$, temos pois que $u =$ P $(o_v + l_v - l_u, l_u)$ e que $vu^{-1} =$ P $(o_v, l_v - l_u)$.

2.6.2 Informações dos vértices

Cada vértice da A^+-árvore pode ser armazenado num vetor de vértices e cada vértice v pode ter algumas informações associadas (por exemplo através de campos de uma estrutura, ou de funções/métodos envolvendo estas informações). Estas informações incluem informações relativas à própria estrutura da árvore tais como o *primeiro filho* do vértice v, denotado por P (v), o *próximo irmão* I ˜ (v), e o *pai*[3] P (v) do vértice v. Convém observar que o conjunto dos filhos de v é dado pela lista ligada

$$\mathrm{P}\quad(v), \mathrm{I}\quad\tilde{}\;(\mathrm{P}\quad(v)), \mathrm{I}\quad\tilde{}\;(\mathrm{I}\quad\tilde{}\;(\mathrm{P}\quad(v))), \ldots, \quad .$$

Como vimos que não há mais que $|A|$ filhos, estas listas ligadas tem no máximo comprimento $|A|$ e o custo de percorrer estas listas e/ou selecionar um filho de um nó será considerado constante (de fato $O(|A|)$).

Interessa-nos incluir também informações relativas aos rótulos dos vértices. Assim o rótulo de um vértice será representado por uma *ocorrência do rótulo* como fator de w, O (v), e o *comprimento do rótulo* C (v). Observe-se que o rótulo de uma aresta (u, v) é

$$\lambda(u)^{-1}\lambda(v) = \mathrm{P}\qquad(\mathrm{O}\quad(v) + \mathrm{C}\quad(u), \mathrm{C}\quad(v) - \mathrm{C}\quad(u)).$$

Além destas informações, veremos na seção 2.7 que será extremamente útil a da *ligação de sufixo de v*, denotada por S (v) e que será então definida. São ainda muitas vezes armazenadas informações que permitam inferir se o vértice dado é final ou não. Sendo final, interessa-nos também qual a posição i tal que o rótulo do vértice seja o sufixo $w[i\,..\,|w|]$.

2.6.3 Análise de complexidade de Á D S

Nesta seção, faremos uma análise da complexidade computacional da função Á D S do Algoritmo 2.3. Observe que Á D S chama, à linha 5, a função A P do Algoritmo 2.2. Esta por sua vez, chama, à linha 9, a função S do Algoritmo 2.1. Assim, precisamos examinar a complexidade computacional de cada uma destas funções. Como dito anteriormente, estaremos sempre representando os fatores de w através de suas ocorrências em w, e seus comprimentos.

[3]De fato, pode-se eliminar a necessidade de guardar a informação relativa ao pai de um vértice, como veremos.

Ademais, seguindo o procedimento já adotado em trabalhos anteriores, supomos que o alfabeto é fixo e seu tamanho finito.

A chamada da função S $(s, 1, \mathbf{T})$ é executada em tempo $O(|h|)$, onde h é a cabeça de s em $\mathbf{T}$ (certamente que $|h| \leq |s|$). (No caso geral, S $(s, r, \mathbf{T})$ é executada em tempo proporcional ao comprimento do prefixo mais comprido de s que pode ser soletrado a partir de r.) De fato, todas as operações são realizadas em tempo constante, exceto as comentadas a seguir. À linha 10, a seleção da aresta $e = (u, v)$ tal que $s[1] \in \mathrm{Pref}(\lambda(e))$ requer tempo constante $O(|A|)$. À linha 11, a identificação do mais comprido prefixo x de $\lambda(e)$ e de s requer tempo $O(|x|)$. À linha 7, a seleção da aresta $e = (u, v)$ tal que $\lambda(e) = \lambda(u)^{-1}\lambda(v) \in \mathrm{Pref}(s)$ requer a seleção de um filho v de u tal que o rótulo da aresta $\lambda(u)^{-1}\lambda(v)$ comece com a letra $s[1]$, feita em tempo constante $O(|A|)$, seguida de uma posterior verificação de cada uma das demais $|\lambda(u)^{-1}\lambda(v)| - 1$ letras do rótulo da aresta. Isto requer tempo $O(|\lambda(e)|)$, sendo que $|\lambda(e)|$ é o acréscimo do comprimento de $\lambda(u)$ à linha 9. Assim, à linha 15, o tempo total de execução da função é claramente limitado por uma função em $O(|\lambda(u)x|)$, sendo que $\lambda(u)x$ é a cabeça de s em $\mathbf{T}$. No Algoritmo 2.4, vemos uma reescritura mais detalhada (e não tão conceitual) da mesma função S .

Outra importante reescritura da função S é a função R S exibida no Algoritmo 2.5. Nesta implementação, impomos a restrição de que $\lambda(r)w$ seja palavra de $\mathbf{T}$, ou seja, sabemos ser possivel soletrar w a partir de r na A^+-árvore $\mathbf{T}$. De posse desta condição, sendo (u, x) o lugar obtido à linha 16 da chamada da função R S $(w, r, \mathbf{T})$, é imediato verificar que o número de iterações do laço à linha 9 é igual ao número de arestas cujos rótulos foram completamente soletrados à linha 12 mais, no caso de $x \neq 1$, a aresta associada ao lugar (u, x) e cujo rótulo foi parcialmente utilizado para soletrar x à linha 15. O tempo de execução, neste caso, é $O(|p|)$ onde p é o caminho de r a u. A função R S não pode ser utilizada em substituição à chamada de S à linha 9 do Algoritmo 2.2, mas será importante como parte da elaboração de um algoritmo linear para a construção da árvore dos sufixos.

A chamada da função A P $(\mathbf{T}, s)$ por sua vez é executada em tempo $O(|h|)$, onde h é a cabeça de s em $\mathbf{T}$ (certamente que $|h| \leq |s|$). De fato, a função A P não possui iterações e todas as operações da função são realizáveis em tempo constante (no máximo $O(|A|)$), exceto a chamada da função S à linha 9, que é realizada em tempo $O(|h|)$.

É imediato verificar que todas as operações realizadas na função Á -

Algoritmo 2.4 Reescritura da função S

```
S (w, r, T)
▹ w: palavra a ser soletrada a partir do
▹ r: vértice de origem, tipicamente, a raiz
▹ T: a A⁺-árvore em questão
▹ devolve a cabeça de λ(r)w em T, seu lugar e aresta associada
u ← r
s ← w
enquanto |s| > 0 faça
    ▹ procura aresta (u, v) tal que seu rótulo começa com s[1]
    v ← P (u)
    enquanto v ≠ faça                ▹ percorre a lista dos filhos
        y ← λ(u)⁻¹λ(v)               ▹ y = λ((u, v))
        se y[1] = s[1] então
            caia fora                ▹ achou tal aresta (u, v)
        v ← I ˜ (v)
    se v = então
        caia fora                    ▹ não existe tal aresta (u, v)
    ▹ o rótulo y da aresta (u, v) começa com a mesma letra que s
    ▹ testa se y ∈ Pref(s)
    i ← 2
    enquanto i ≤ |y| e i ≤ |s| e y[i] = s[i] faça
        i ← i + 1
    se i > |y| então                 ▹ y ∈ Pref(s)
        s ← y⁻¹s
        u ← v
    senão x ← y[1 . . i − 1]         ▹ y ∉ Pref(s)
        caia fora
se |s| = 0 ou v = então
    x ← 1
    v ← u                            ▹ o lugar não está numa aresta
devolva (λ(u)x, u, x, v)             ▹ λ(u)x é cabeça de λ(r)w em T
```

Algoritmo 2.5 Função R S

R S $(w, r, \mathbf{T})$
▹ w: palavra a ser soletrada a partir do
▹ r: vértice de origem, tipicamente, a raiz
▹ $\mathbf{T}$: a A^+-árvore em questão
▹ Restrição importante: $\lambda(r)w$ é palavra de $\mathbf{T}$
▹ devolve a cabeça de $\lambda(r)w$ em $\mathbf{T}$, seu lugar e aresta associada
$u \leftarrow r$
$s \leftarrow w$
$x \leftarrow 1$
enquanto $|s| > 0$ **faça**
    $(u, v) \leftarrow$ aresta $e \mid s[1] \in \text{Pref}(\lambda(e))$ ▹ $\lambda(e) = \lambda(u)^{-1}\lambda(v)$
    se $|s| \geq |\lambda(e)|$ **então**
        $s \leftarrow \lambda(e)^{-1}s$
        $u \leftarrow v$
    senão $x \leftarrow s$
        $s \leftarrow 1$
devolva $(\lambda(u)x, u, x, v)$

D S no Algoritmo 2.3 são feitas em tempo constante, a não ser talvez pelas chamadas à função A P . Como a iteração da linha 5 é executada $n + 1$ vezes, é imediato verificar que a construção da árvore dos sufixos de w feita por Á D S é $O(n)$ mais o tempo gasto em cada uma das execuções de A P . Cada uma destas execuções da função A P do Algoritmo 2.2, que adiciona um sufixo $w[i \,..\, n]$, é $O(n - i + 1)$, como vimos. Isto resulta numa complexidade final quadrática. De fato, esta complexidade final é atingida no exemplo $w = a^m b$. Para $k = m-1, m-2, \ldots 0$, a cabeça de cada sufixo $a^k b$ na A^+-árvore de reconhecimento de $\cup_{i=k+1}^{m}\{a^i b\}$ é a^k. Assim, o tempo dispendido com as sucessivas chamadas à função S feitas à linha 9 do Algoritmo 2.2 é aproximandamente proporcional à soma

$$\sum_{k=0}^{m-1} |a^k| = \frac{m(m-1)}{2}.$$

Na Figura 2.6 vemos a A^+-árvore de reconhecimento de $a^4 b$. Os vértices são inseridos à A^+-árvore $\mathbf{T}$ da função Á D S na ordem 1, $a^4 b$, a^3,

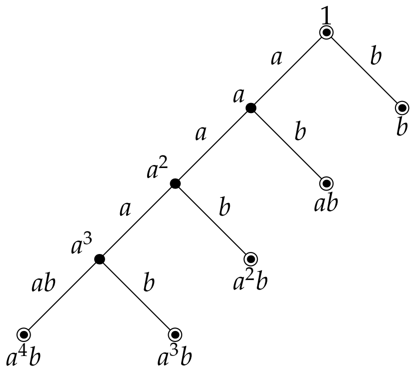


Figura 2.6: A^+-árvore de reconhecimento de a^4b

a^3b, a^2, a^2b, a, ab, b. As sucessivas cabeças soletradas pela função S são a^3, a^2, a, 1, 1.

2.7 Ligação de sufixo

A análise feita à Seção 2.6.3 mostra o quão crítica é a computação das cabeças h_i feitas à linha 9 do Algoritmo 2.2. Nesta seção, cuidaremos de mostrar como estas computações podem ser feitas em tempo amortizado constante. Para tanto, será fundamental a manutenção das ligações de sufixo S (v), definidas a seguir.

Para $i = 0, 1, \dots, n+1$, também definimos a seqüência de conjuntos $W_i \subseteq \mathrm{Suf}(w)$, e a seqüencia de A^+-árvores $\mathbf{T}_i$ como sendo

$$
\begin{aligned}
W_i &= \cup_{k=1}^{i}\{w[k\,..\,n]\} = \begin{cases} \emptyset, & \text{se } i = 0, \\ W_{i-1} \cup \{w[i\,..\,n]\}, & \text{se } i > 0, \end{cases} \\
\mathbf{T}_i &= A^+\text{-árvore de reconhecimento de } W_i.
\end{aligned}
$$

(Observe que $w[n+1\,..\,n] = 1$.) Temos então que $\mathbf{T}_{n+1}$ é a A^+-árvore de reconhecimento de $W_{n+1} = \mathrm{Suf}(w)$, a árvore dos sufixos de w. Por fim, para $i = 2, \dots, n+1$, definimos h_i e t_i como sendo a cabeça e a cauda do sufixo $w[i\,..\,n]$ em $\mathbf{T}_{i-1}$, definimos (u_i, x_i) como sendo o lugar de h_i em $\mathbf{T}_{i-1}$, e v_i como sendo o término da aresta associada a este lugar, quando o lugar está numa aresta. Devido ao Lema 2.6, temos que h_i e $w[i\,..\,n]$ são vértices (iguais se e só se $w[i\,..\,n] \in \mathrm{Pref}(W_{i-1})$) de $\mathbf{T}_i$, para $i = 2, \dots, n+1$. Ademais, os demais vértices já ocorrem em $\mathbf{T}_{i-1}$.

Logo a seguir, no Lema 2.9, provaremos uma importante propriedade mostrando que o mais longo prefixo próprio de h_{i-1}, $A^{-1}h_{i-1}$, é prefixo de h_i. Esta propriedade motiva a seguinte definição. Dada uma A^+-árvore $\mathbf{T}$ e dois vértices u e v tais que existe letra a tal que $\lambda(u) = a\lambda(v)$, definimos a *ligação de sufixo de u*, denotada por S (u), como sendo o vértice v. (De fato, $A^{-1}h_{i-1}$ é a eventual ligação de sufixo de h_{i-1}.) No caso da raiz, definimos sua ligação de sufixo por S (R ($\mathbf{T}$)) = , que não é um vértice de $\mathbf{T}$, e não definimos a ligação de sufixo de vértices u tais que não haja vértice de rótulo $A^{-1}\lambda(u)$. Devido à Proposição 2.7, dado um vértice u diferente da raiz, sua ligação de sufixos v é um vértice de $\mathbf{T}$ se a árvore em questão for A^+-árvore de reconhecimento de um conjunto fechado por sufixos, como é o caso de uma árvore dos sufixos. Na Figura 2.7, vemos uma ilustração que destaca dois vértices au e u de uma A^+-árvore de reconhecimento e a ligação de sufixo de au para u. As ligações de sufixo serão sempre representadas por arestas tracejadas, como nesta figura.

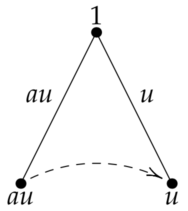


Figura 2.7: Ligação de sufixo u = S (au)

Lema 2.9 *Suponha que $3 \leq i \leq n$ e que $|h_{i-1}| > 0$. Então a palavra $A^{-1}h_{i-1}$ é um prefixo de h_i. Ademais, se este prefixo é próprio temos que $A^{-1}h_{i-1}$ é um vértice de $\mathbf{T}_{i-1}$.*

Prova. Devido ao item 5 do Teorema 2.4 e às definições de h_k e $\mathbf{T}_{k-1}$, temos que uma cabeça h_k é o mais longo prefixo de $w[k\,..\,n]$ que é prefixo de W_{k-1}, para $k = 2, \ldots, n+1$. Como $h_{i-1} \in \mathrm{Pref}(W_{i-2}) = \mathrm{Pref}(\cup_{k=1}^{i-2}\{w[k\,..\,n]\})$, seja pois $j \in \{1, \ldots, i-2\}$ tal que $h_{i-1} \in \mathrm{Pref}(w[j\,..\,n])$.

Vamos provar que $A^{-1}h_{i-1} \in \mathrm{Pref}(h_i)$. Como $h_{i-1} \in \mathrm{Pref}(w[j\,..\,n])$, temos que $A^{-1}h_{i-1} \in \mathrm{Pref}(A^{-1}w[j\,..\,n]) = \mathrm{Pref}(w[j+1\,..\,n]) \subseteq \mathrm{Pref}(W_{i-1})$. Como $A^{-1}h_{i-1} \in \mathrm{Pref}(A^{-1}w[i-1\,..\,n]) = \mathrm{Pref}(w[i\,..\,n])$, temos pois que $|h_i| \geq |A^{-1}h_{i-1}|$ já que h_i é o prefixo mais comprido de $w[i\,..\,n]$ que é prefixo de W_{i-1}. Como $h_i, A^{-1}h_{i-1} \in \mathrm{Pref}(w[i\,..\,n])$, temos por fim que $A^{-1}h_{i-1} \in \mathrm{Pref}(h_i)$.

Suponha que $A^{-1}h_{i-1}$ seja prefixo próprio de h_i. Vamos provar que $A^{-1}h_{i-1}$ é uma bifurcação de W_{i-1}, e portanto, um vértice de $\mathbf{T}_{i-1}$. Por um lado, como $A^{-1}h_{i-1}t_{i-1} = A^{-1}w[i-1\,..\,n] = w[i\,..\,n] = h_it_i$, como $A^{-1}h_{i-1}$ é prefixo próprio de h_i, temos que $|t_{i-1}| = |A^{-1}h_{i-1}t_{i-1}| - |A^{-1}h_{i-1}| = |h_it_i| - |A^{-1}h_{i-1}| \geq |h_i| - |A^{-1}h_{i-1}| \geq 1$. Seja pois a a primeira letra de t_{i-1}. Assim, $A^{-1}h_{i-1}a, h_i \in \operatorname{Pref}(A^{-1}h_{i-1}t_{i-1}) = \operatorname{Pref}(h_it_i)$. Como $|h_i| \geq |A^{-1}h_{i-1}| + 1 = |A^{-1}h_{i-1}a|$, segue que $A^{-1}h_{i-1}a \in \operatorname{Pref}(h_i) \subseteq \operatorname{Pref}(W_{i-1})$. Por outro lado, $h_{i-1} \in \operatorname{Pref}(w[i-1\,..\,n])$ implica que $|h_{i-1}| \leq |w[i-1\,..\,n]| < |w[j\,..\,n]|$. Como $h_{i-1} \in \operatorname{Pref}(w[j\,..\,n])$, podemos definir b a primeira letra de $h_{i-1}^{-1}w[j\,..\,n]$. Assim, temos que $h_{i-1}b \in \operatorname{Pref}(w[j\,..\,n]) \subseteq \operatorname{Pref}(W_{i-2})$ e que $A^{-1}h_{i-1}b \in \operatorname{Pref}(w[j+1\,..\,n]) \subseteq \operatorname{Pref}(W_{i-1})$. Por um terceiro lado, como $h_{i-1}a \in \operatorname{Pref}(h_{i-1}t_{i-1}) = \operatorname{Pref}(w[i-1\,..\,n])$, como $h_{i-1}b \in \operatorname{Pref}(W_{i-2})$, como h_{i-1} é o mais longo prefixo de $w[i-1\,..\,n]$ que é prefixo de W_{i-2}, segue que $a \neq b$. Por fim, $A^{-1}h_{i-1}a, A^{-1}h_{i-1}b \in \operatorname{Pref}(W_{i-1})$ com $a \neq b$, o que conclui a prova de que $A^{-1}h_{i-1}$ é uma bifurcação de W_{i-1}. ∎

Como vimos, a ligação de sufixo de um vértice u sempre existe se a árvore em questão for A^+-árvore de reconhecimento de um conjunto fechado por sufixos, como é o caso de uma árvore dos sufixos. Contudo, isto não ocorre, em geral, com as árvores $\mathbf{T}_i$. Em contrapartida, os únicos vértices de $\mathbf{T}_{i-1}$ que eventualmente podem não ter definidas suas ligações de sufixo são os vértices não presentes a $\mathbf{T}_{i-2}$, na pior das hipóteses, os dois vértices h_{i-1} e $w[i-1\,..\,n]$. A partir de agora, suporemos que as ligações de sufixo dos vértices de $\mathbf{T}_{i-2}$, exceto a raiz, são sempre definidas e conhecidas em tempo $O(1)$ quando queremos computar a árvore $\mathbf{T}_i$. Devido ao Lema 2.9, de posse da ligação de sufixo de um vértice h_{i-1}, poderíamos sempre acelerar o cálculo da próxima cabeça h_i substituindo as sucessivas chamadas S $(w[i\,..\,n], 1, \mathbf{T})$ feitas à linha 9 do Algoritmo 2.2 por S $(t_{i-1}, \text{S}\ (h_{i-1}), \mathbf{T})$. (Observe que $t_{i-1} = \lambda(\text{S}\ (h_{i-1}))^{-1}w[i\,..\,n]$.) Como veremos na Seção 2.8, o tempo total das computações dispendidas com estas chamadas é $O(n)$. Se por um lado tal aceleração não é ainda tão simples devido à não definição de S (h_{i-1}) quando h_{i-1} não é vértice de $\mathbf{T}_{i-2}$, tal problema não ocorre com os vértices já presentes em $\mathbf{T}_{i-2}$, inclusive P (h_{i-1}), se h_{i-1} não for a raiz. Suponhamos o caso mais complexo que ocorre quando nem h_{i-1} nem seu pai sejam a raiz. Assim, existem $a \in A$ e $u, x \in A^*$ tais que P $(h_{i-1}) = au$ e $h_{i-1} = aux$. Assim, temos que S $(\text{P}\ (h_{i-1})) = u$ é definida, e o Lema 2.9 garante-nos que $ux \in \operatorname{Pref}(h_i)$. Seja $y \in A^*$ tal que $h_i = uxy$. Na Figura 2.8 vemos uma ilustração desta situação, para um caso em que $y \neq 1$. Podemos calcular a cabeça h_i através

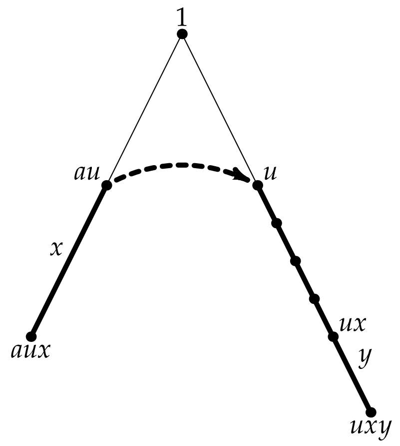


Figura 2.8: Cálculo de cabeça usando ligação de sufixo

da chamada S $(u^{-1}w[i\,..\,n], u, \mathbf{T})$. Observe que $u^{-1}w[i\,..\,n] = xt_{i-1}$ e que esta computação demora tempo $O(|xy|)$. Dado que o Lema 2.9 nos garante que $ux \in \mathrm{Pref}(h_i)$, isto pode ser acelerado através da computação do lugar de ux através de R S $(x, u, \mathbf{T})$, seguida de posterior chamada de S $(t_{i-1}, ux, \mathbf{T})$, se $y \neq 1$. De fato, se $y \neq 1$, o próprio Lema 2.9 garante-nos que ux é um vértice (que também deve ser avaliado como S (h_{i-1})) e a computação S $(t_{i-1}, ux, \mathbf{T})$ é definida.

No Algoritmo 2.6 temos um algoritmo geral para a computação de h_i em função de h_{i-1} e de seu lugar (u, x) em $\mathbf{T}_{i-2}$. A função N C usa a ligação de sufixo e a função R S sempre que possível e usa a função S quando não há alternativa. O caso acima comentado, o da Figura 2.8, leva à execução das linhas 16 e 19. Caso tivéssemos $y = 1$ e $h_i = A^{-1}h_{i-1} = ux$, teríamos a execução das mesmas linhas se o lugar de ux fosse um vértice, ou das linhas 16 e 20 se ux não fosse um vértice. O caso em que o pai u de h_{-1} é a raiz, é tratado de forma especial com a chamada de R S $(A^{-1}x, 1, \mathbf{T}_{i-1})$ à linha 14 ao invés de R S $(x, \mathrm{S}\ (u), \mathbf{T}_{i-1})$ à linha 16. Naturalmente estes cálculos todos na tentativa de usar a ligação de sufixo do pai de h_{i-1} só se fazem necessários quando h_{i-1} não é um vértice de $\mathbf{T}_{i-2}$ e sua ligação de sufixo não é definida. O caso mais simples em que h_{i-1} é um vértice de $\mathbf{T}_{i-2}$, leva à execução de S $(t_{i-1}, \mathrm{S}\ (h_{i-1}), \mathbf{T}_{i-1})$ à linha 10 se $h_{i-1} \neq 1$ e à execução de S $(A^{-1}t_{i-1}, 1, \mathbf{T}_{i-1})$ à linha 8 se $h_{i-1} = 1$.

Algoritmo 2.6 Cálculo de h_i usando ligação de sufixo

N C ¸ $(u_{i-1}, x_{i-1}, \mathbf{T}_{i-1})$
▹ (u_{i-1}, x_{i-1}): lugar de h_{i-1} em $\mathbf{T}_{i-2}$
▹ Observe que h_{i-1} é vértice de $\mathbf{T}_{i-2}$ se e só se $x_{i-1} = 1$
▹ Observe que $u_{i-1} =$ P (h_{i-1}) quando $x_{i-1} \neq 1$
se $x_{i-1} = 1$ **então**
    ▹ $u_{i-1} = h_{i-1}$ é vértice de $\mathbf{T}_{i-2}$
    se $u_{i-1} = 1$ **então**
        ▹ h_{i-1} é a raiz e S $(h_{i-1}) =$
        $(h_i, u_i, x_i, v_i) \leftarrow$ S $(A^{-1}t_{i-1}, 1, \mathbf{T}_{i-1})$
    senão ▹ S (h_{i-1}) é vértice
        $(h_i, u_i, x_i, v_i) \leftarrow$ S $(t_{i-1},$ S $(u_{i-1}), \mathbf{T}_{i-1})$
senão ▹ h_{i-1} não é vértice de $\mathbf{T}_{i-2}$ mas $u_{i-1} =$ P (h_{i-1}) é
    se $u_{i-1} = 1$ **então**
        ▹ u_{i-1} é a raiz e S $(u_{i-1}) =$
        $(h, u, x, v) \leftarrow$ R S $(A^{-1}x_{i-1}, 1, \mathbf{T}_{i-1})$
    senão ▹ S (u_{i-1}) é vértice
        $(h, u, x, v) \leftarrow$ R S $(x_{i-1},$ S $(u_{i-1}), \mathbf{T}_{i-1})$
    se $x = 1$ **então** ▹ $h = A^{-1}h_{i-1} = u$ é vértice
        S $(h_{i-1}) \leftarrow u$
        $(h_i, u_i, x_i, v_i) \leftarrow$ S $(t_{i-1}, u, \mathbf{T}_{i-1})$
    senão $(h_i, u_i, x_i, v_i) \leftarrow (h, u, x, v)$
devolva (h_i, u_i, x_i, v_i) ▹ (u_i, x_i): lugar de h_i em $\mathbf{T}_{i-1}$

Na Figura 2.9 vemos a A^+-árvore de reconhecimento de $w = a^5b^2aba^3ba^4$ e uma ilustração do uso de ligação de sufixo para a computação de uma cabeça. Nesta árvore, os dois últimos vértices inseridos (na árvore $\mathbf{T}_{13}$) são

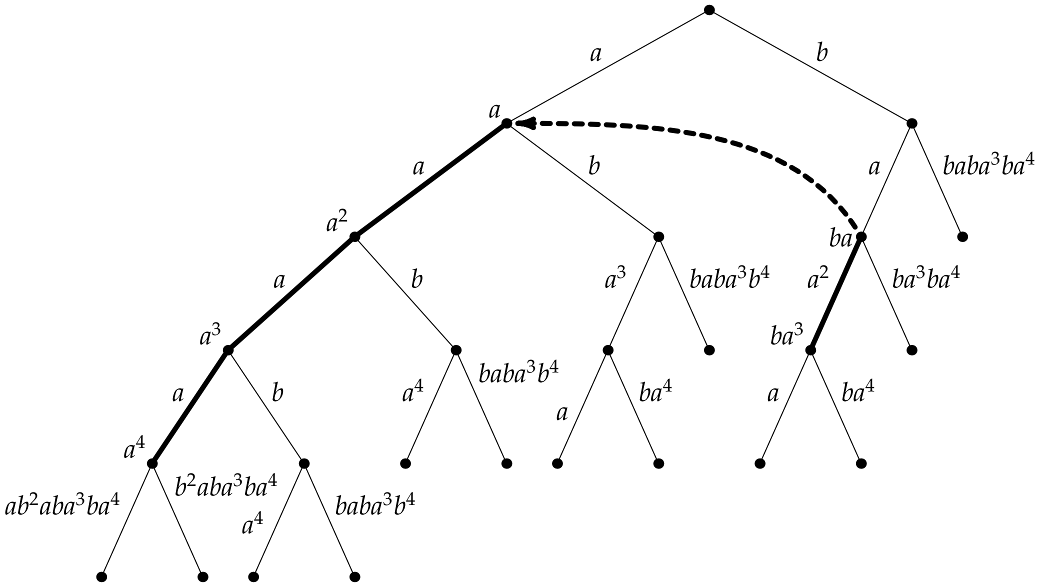


Figura 2.9: Cálculo da cabeça h_{14} de a^4 na Á D S ($a^5b^2aba^3ba^4$) usando ligação de sufixo de P (h_{13}) = P (ba^3) = ba para a

o sufixo $ba^4 = w[13\,..\,17]$ e sua cabeça $h_{13} = ba^3$. O lugar de ba^3 na árvore $\mathbf{T}_{12}$ é $(u, x) = (ba, a^2)$. O próximo sufixo a ser inserido à árvore é $a^4 = w[14\,..\,17]$, que é palavra de $\mathbf{T}_{13}$. Para a computação da cabeça h_{14} de a^4 em $\mathbf{T}_{13}$, partimos do vértice $h_{13} = ba^3$. Já dispomos de seu pai $u =$ P $(h_{13}) = ba$, fazemos uso da ligação de sufixo S $(ba) = a$ e soletramos $x = a^2$ a partir de a usando a função R S . Como o lugar encontrado é de um vértice, a^3, soletramos então $t_{13} = a$ usando a função S . Assim, encontramos $h_{14} = w[14\,..\,17] = a^4$. (Aliás, $h_i = w[i\,..\,17] = a^{18-i}$, para $i = 14, 15, 16, 17, 18$, são vértices de $\mathbf{T}_{13}$.)

Na Figura 2.8, temos vértices internos no caminho de u a ux, apesar de não haver vértice interno algum no caminho de au para aux. Situações como estas são realistas como vemos na Figura 2.9, onde há o vértice a^2 no caminho de a para a^3 mas não há vértice interno no caminho de ba para ba^3. Esta situação merecerá cuidados especiais na próxima seção.

2.8 Construção linear da árvore dos sufixos

Nesta seção apresentamos o algoritmo final da construção da árvore dos sufixos e damos uma análise de eficiência do mesmo. Este algoritmo (Algoritmo 2.7) faz uso das ligações de sufixos de forma a permitir uma construção em tempo linear no comprimento da palavra w, como veremos. Fundamentalmente, o algoritmo funde as funções Á D S -, A P e N C ¸ vistos nos Algoritmos 2.3, 2.2 e 2.6 numa nova função Á D S . Aproveitamos a ocasião para já apresentar uma solução de um problema mais geral, que constrói uma A^+-árvore que reconheça os sufixos de várias palavras.

A função Á D S vista no Algoritmo 2.7 é completa, mas alguns detalhes foram omitidos por razão de simplicidade. Junto às inicializações, definimos S (1) = . Devemos garantir que os vértices de $\mathbf{T}_{i-2}$ tenham suas ligações de sufixo sempre calculadas. Isto é feito à linha 30 para os casos $i \geq 2$. Caso uma das linhas 18 ou 17 tenham sido executadas na iteração $i \geq 2$, deve-se definir S (h_{i-1}) como sendo o valor de u antes de alteração à linha 20 se a mesma foi executada ou o vértice h criado à linha 23 caso contrário. Caso nenhuma das linhas 18 ou 17 tenham sido executadas, S (h_{i-1}) já foi definida anteriormente. Ademais, S $(w[i-1\,..\,n])$ deve ser definida como sendo $w[i\,..\,n]$ (a folha terá sido eventualmente inserida à linha 28).

Vale observar que as ocorrências dos rótulos dos vértices O (v) como fatores de w não se referem mais a uma única palavra w, dado que há várias $w \in W$ sendo usadas no algoritmo. Isto constitui um problema e pode-se adotar uma das três seguintes soluções: supondo-se que todas as palavras de W sejam fatores de uma mesma palavra w', podemos adotar as ocorrências do fatores em questão como fatores de w'; pode-se adotar ainda o endereço de memória (algo como `&w[ocorrencia]` em C) do referido fator; pode-se adotar um par de informações que identifique a ocorrência em w e a particular palavra w em questão.

Vamos agora fazer uma análise da complexidade computacional do Algoritmo 2.7. Por simplicidade, suporemos que $W = \{w\}$ contém uma única palavra w. Suporemos as definições de h_i, t_i, u_i, x_i, W_i e $\mathbf{T}_i$ praticamente idênticas às feitas à Seção 2.7. A principal diferença se dá à definição de $W_0 = \{1\}$ e da definição de h_i, t_i, (u_i, x_i) e v_i também para $i = 1$. Esta ligeira variação não introduz qualquer alteração quando $i \geq 2$, mas se deve à linha 4 do Algoritmo 2.7, cuja introdução simplifica a elabora-

Algoritmo 2.7 Construção de árvore dos sufixos em tempo linear

Á D S (W)

$\triangleright$ W: conjunto de palavras cuja árvore dos sufixos será calculada
T $\leftarrow$ árvore trivial
se $W = \emptyset$ **então**
  devolva T
acrescente uma raiz 1 $\triangleright$ $\mathbf{T}_0$
para cada $w \in W$ **faça**
  $n \leftarrow |w|$
  para i **de** 1 **até** $n + 1$ **faça**
    se $i = 1$ **então**
      $(h, u, x, v) \leftarrow$ S $(w, 1, \mathbf{T})$
    senão-se $x = 1$ **então** $\triangleright$ $u = h_{i-1}$ é vértice de $\mathbf{T}_{i-2}$
      se $u = 1$ **então** $\triangleright$ S $(h_{i-1}) =$
        $(h, u, x, v) \leftarrow$ S $(A^{-1}t, 1, \mathbf{T})$
      senão $(h, u, x, v) \leftarrow$ S $(t,$ S $(u), \mathbf{T})$
    senão $\triangleright$ h_{i-1} não é vértice de $\mathbf{T}_{i-2}$ mas $u =$ P (h_{i-1}) é
      se $u = 1$ **então** $\triangleright$ S (P $(h_{i-1})) =$
        $(h, u, x, v) \leftarrow$ R S $(A^{-1}x, 1, \mathbf{T})$
      senão $(h, u, x, v) \leftarrow$ R S $(x,$ S $(u), \mathbf{T})$
      se $x = 1$ **então** $\triangleright$ $h = A^{-1}h_{i-1} = u$ é vértice
        $(h, u, x, v) \leftarrow$ S $(t, u, \mathbf{T})$
    $t \leftarrow h^{-1}w$
    se $|x| > 0$ **então**
      acrescente novo vértice h
      remova a aresta (u, v)
      acrescente aresta de u para h de rótulo x
      acrescente aresta de h para v de rótulo $h^{-1}v$
    se $|t| > 0$ **então**
      acrescente nova folha $w[i \,..\, n]$
      acrescente aresta de h para $w[i \,..\, n]$ de rótulo t
    atualize S (h_{i-1}) e S $(w[i - 1 \,..\, n])$
devolva T

ção do algoritmo para o conjunto de palavras W. Para $i = 0, 1, \ldots, n+1$, também definimos a seqüência de conjuntos $W_i \subseteq \mathrm{Suf}(w)$, e a seqüencia de A^+-árvores $\mathbf{T}_i$ como sendo

$$\begin{aligned} W_i &= \{1\} \cup \cup_{k=1}^{i} \{w[k\,..\,n]\} = \begin{cases} \{1\}, & \text{se } i = 0, \\ W_{i-1} \cup \{w[i\,..\,n]\}, & \text{se } i > 0, \end{cases} \\ \mathbf{T}_i &= A^+\text{-árvore de reconhecimento de } W_i. \end{aligned}$$

Para $i = 1, \ldots, n+1$, definimos h_i e t_i como sendo a cabeça e a cauda do sufixo $w[i\,..\,n]$ em $\mathbf{T}_{i-1}$, definimos (u_i, x_i) como sendo o lugar de h_i em $\mathbf{T}_{i-1}$, e v_i como sendo o término da aresta associada a este lugar, quando o lugar está numa aresta. Devido ao Lema 2.6, temos que h_i e $w[i\,..\,n]$ são vértices de $\mathbf{T}_i$, para $i = 1, \ldots, n+1$. Ademais, os demais vértices já ocorrem em $\mathbf{T}_{i-1}$. Definamos ainda $h_0 = 1$ e $t_0 = w$.

Já fizemos à Seção 2.6.3 uma análise dos algoritmos anteriormente vistos e não repetiremos aqui a mesma argumentação feita anteriormente. A parte crítica daqueles algoritmos, que levava a um tempo de execução quadrático era o cálculo das cabeças, que no Algoritmo 2.7 é feito entre as linhas 10 e 21. Observe-se que uma eventual chamada à função S na iteração i é feita na tentativa de soletrar t_{i-1} (ou eventualmente seu sufixo próprio mais comprido $A^{-1}t_{i-1}$) e que o tempo de processamento é proporcional ao comprimento do prefixo de t_{i-1} (eventualmente, de $A^{-1}t_{i-1}$) que foi soletrado, a saber, $t_{i-1}t_i^{-1}$ (eventualmente, $A^{-1}t_{i-1}t_i^{-1}$). Ao final de todas as iterações, o tempo será proporcional a

$$\sum_{i=1}^{n+1} |t_{i-1}t_i^{-1}| = |t_0| - |t_{n+1}| = |w| - 0 = |w| = n.$$

Resta-nos apenas analisar o tempo dispendido com as chamadas à função R S . Para cada iteração i, seja k_i o número de arestas visitadas durante a eventual chamada da função R S (chamada de R S $(x_{i-1}, \mathrm{S}(u_{i-1}), \mathbf{T}_{i-1})$ à linha 18 ou de R S $(A^{-1}x_{i-1}, 1, \mathbf{T}_{i-1})$ à linha 17 ou nenhuma chamada de R S). Seja $\varphi(i)$ o comprimento da palavra que resta ser soletrada com R S e S na iteração i (nos dois primeiros casos acima temos $\varphi(i) = |x_{i-1}t_{i-1}|$ ou $\varphi(i) = |A^{-1}x_{i-1}t_{i-1}|$ e no terceiro podemos ter $\varphi(i) = |t_{i-1}|$ ou $\varphi(i) = |A^{-1}t_{i-1}|$). Eventualmente, o lugar de h_i está numa aresta e o rótulo da última das k_i arestas deverá ser novamente resoletrado por R S na iteração $i+1$.

Contudo, em qualquer caso, os rótulos das demais $k_i - 1$ arestas terão sido definitivamente soletrados. Como o rótulo de cada aresta visitada é não vazio, temos então que $\varphi(i+1) \leq \varphi(i) - (k_i - 1)$ e que

$$k_i \leq \varphi(i) - \varphi(i+1) + 1.$$

Assim, o tempo total dispendido com as chamadas à função R S é proporcional a

$$\begin{aligned} \sum_{i=1}^{n+1} k_i &\leq \sum_{i=1}^{n+1} (\varphi(i) - \varphi(i+1) + 1) \\ &= \varphi(1) - \varphi(i+2) + n + 1 \\ &= |w| - 0 + n + 1 \\ &= 2n + 1. \end{aligned}$$

Assim, temos concluída a demonstração do teorema seguinte.

Teorema 2.10 *O Algoritmo 2.7 constrói a árvore dos sufixos de um conjunto de palavras W em tempo linear à soma dos comprimentos das palavras de W.*

Convém observar que, além da árvore de sufixos propriamente dita, muitas vezes interessa-nos também construir a tabela V´ F (w, i) que permite inferir o vértice final cujo rótulo é o sufixo $w[i\,..\,|w|]$ em tempo $O(1)$. Ela pode ser construída com atualizações iterativas feitas à linha 30 do Algoritmo 2.7. Deseja-se também poder inverter esta tabela e dado um vértice final v deseja-se muitas vezes a ocorrência do sufixo (ou as ocorrências dos eventuais sufixos no caso de W ter mais que uma palavra) que é o rótulo de v. No caso de W somente possuir uma palavra, esta informação pode ser já dada pela ocorrência O (v).

2.9 Notas Bibliográficas

Um importante tópico a ser considerado quando se procura uma implementação eficiente de uma construção de uma árvore de sufixos, é a razão entre a quantidade de memória utilizada pelas estruturas de dados e o comprimento da palavra cuja árvore de sufixos está sendo construída. Uma discussão adequada do assunto foge ao escopo do presente trabalho.

Convém porém observar que após a publicação do primeiro algoritmo linear de construção da árvore de sufixos de uma palavra ser desenvolvido por Weiner [Wei73], McCreight [McC76] apresentou uma construção mais eficiente no uso de memória. A apresentação da função Á D S é de certa forma mais próxima à do trabalho de McCreight.

Ukkonen [Ukk95] desenvolveu a primeira construção linear on-line da árvore dos sufixos. As árvores implícitas construídas pelo algoritmo de Ukkonen são o conceito na literatura que consideramos mais próximo ao de A^+-árvore de reconhecimento por nós desenvolvido.

A construção mais eficiente no uso de memória ainda hoje conhecida é devida a Kurtz [Kur99], e sua implementação é a base do sistema REP [KCO+01] que faz análise de repetições em seqüências. Uma nova versão do alinhador de seqüências MUM [DPCS02] introduziu esta implementação de árvore dos sufixos de forma a ser utilizado na análise completa do genoma humano desenvolvida pela Celera Genomics e publicada recentemente em *Science* [VAM+01].

Não menos importante, há toda uma diferente abordagem ao assunto que desenvolve não uma árvore dos sufixos tal como a definimos, mas uma estrutura prima-irmã da mesma: o autômato dos sufixos. Alguns dos trabalhos mais fundamentais sobre o assunto podem ser encontrados em [BBH+85, Cro86, CV97a, CV97b] e um leitor mais interessado pode aprofundar-se com uma excelente apresentação de Karpischek [Kar93].

Capítulo 3

Aplicações das Árvores dos Sufixos

Neste capítulo, veremos algumas aplicações do uso de árvores dos sufixos na resolução eficiente de alguns problemas sobre seqüências. Alguns destes problemas são de particular interesse à Biologia Molecular Computacional, onde estas soluções eficientes são as únicas aceitáveis face ao tamanho das seqüências genômicas. As seções subseqüentes serão organizadas de acordo com os problemas aos quais os algoritmos propostos pretendem resolver.

Em vários dos problemas a serem vistos, duas seqüências são envolvidas. As duas seqüências em questão são sempre denotadas por s e t e seus respectivos comprimentos são m e n.

3.1 Busca de padrão

Um dos problemas mais básicos nas computações envolvendo palavras é da verificação se uma palavra s de comprimento m, também chamada padrão, é fator de uma outra palavra t de comprimento n, também chamada texto. (Vale dizer que em Biologia Computacional este problema é bastante freqüente e o padrão é muitas vezes chamado de 'motivo'.) O algoritmo ingênuo apresentado no Algoritmo 3.1 resolve o problema com complexidade quadrática $O(m(n-m))$ já que o número de comparações potencialmente realizadas à linha 10 é $|s|(|t|-|s|+1)$ ($|t|-|s|+1$ iterações do laço à linha 6 vezes $|s|$ iterações do laço à linha 9). De fato esta comple-

Algoritmo 3.1 Algoritmo ingênuo de busca de padrão s num texto t

B I ˆ D P ˜ (s, t)

▹ s: palavra s a ser procurada num
▹ t: texto t
▹ devolve $s \in \mathrm{Fat}(t)$ (se $s \in \mathrm{Fat}(t)$ e c.c.)
$r \leftarrow$
$i \leftarrow 0$
enquanto $i \leq |t| - |s|$ **e** $r =$ **faça**
    $r \leftarrow$
    $j \leftarrow 1$ ▹ testa se $s = t[i + 1 \,..\, |s|]$
    enquanto $j \leq |s|$ **e** $r =$ **faça**
        se $t[i + j] \neq s[j]$ **então**
            $r \leftarrow$
        $j \leftarrow j + 1$
    $i \leftarrow i + 1$
devolva r

xidade de pior caso é alcançada para um exemplo como o da procura do padrão $s = a^{m-1}b$ num texto $t = a^n$, com $m \leq n$ (por exemplo, $n = 2m$).

Historicamente, vários algoritmos de busca em tempo linear $O(m + n)$ foram propostos. Para citar alguns dos mais conhecidos, vale citar o desenvolvido por Knuth, Morris e Pratt [KMP77] e por Boyre e Moore [BM77]. Rigorosamente falando, o algoritmo original de Boyre e Moore tinha complexidade de pior caso $O(mn)$, mas pequenas modificações tornaram o algoritmo linear. O grande interesse pela classe de algoritmos originada por Boyre e Moore é o fato do tempo esperado de busca ser sublinear, ou seja menor que $m + n$. Em algumas classes de casos o tempo esperado de execução é $O(m + n/m)$. Estes algoritmos têm por característica pre-processar s para então fazer a busca em t.

Apresentamos no Algoritmo 3.2 outra implementação linear de um algoritmo de busca de padrão s num texto t que faz uso da construção em tempo linear de uma árvore dos sufixos. Neste algoritmo, o preprocessamento é feito sobre o texto t, ao contrário das demais soluções lineares. Seja $\mathbf{T}$ a árvore dos sufixos de t, construída à linha 4 em tempo $O(|t|)$ devido ao Teorema 2.10. Ora, devido ao Corolário 2.8, s é fator de t se e somente se s é palavra de $\mathbf{T}$ se e só se a cabeça de s em $\mathbf{T}$ é a própria palavra s. A cabeça

Algoritmo 3.2 Busca de s em t baseada em árvore dos sufixos

B D P ˜ (s, t)
1 ▷ s: palavra s a ser procurada num
2 ▷ t: texto t
3 ▷ devolve $s \in \mathrm{Fat}(t)$ (se $s \in \mathrm{Fat}(t)$ e c.c.)
4 $\mathbf{T} \leftarrow$ Á D S $(\{t\})$
5 ▷ Calcula a cabeça h de s em $\mathbf{T}$
6 $(h, u, x, v) \leftarrow$ S $(s, 1, \mathbf{T})$
7 ▷ Note que $h = s \iff s \in \mathrm{Fat}(t)$
8 **devolva** $h = s$

h de s em $\mathbf{T}$ é computada à linha 6 e, como vimos à Seção 2.6.3, dispende tempo $O(|s|)$. Daí, o tempo total de computação de B D P ˜ (s, t) é linear $O(m + n)$.

Em muitas aplicações, não somente estamos interessados em saber se s é ou não um fator de t, mas também quais as ocorrências de s como fator de t. Se por um lado a função B D P ˜ no Algoritmo 3.2 não calcula explicitamente estas posições em t, por outro lado já o faz implicitamente. De fato, estas ocorrências correspondem ao valores de V´ F (x), para todo vértice final x que está na sub-árvore de raiz v, onde o vértice v é computado à linha 6. Uma variante que tivesse interesse em devolver também estas ocorrências poderia devolver v.

Vale comentar que algumas generalizações do problema são bastante úteis e freqüentes. Uma delas é a feita por Aho e Corasick [AC75] que procura uma das palavras de um conjunto finito de padrões S num texto t. O tempo total é linear $O(m+n)$ onde m, no caso, é a soma dos comprimentos das palavras de S.

Outra generalização freqüentemente encontrada é aquela que procura um fator do texto que pertença a uma linguagem regular dada. Neste caso os algoritmos conhecidos supõem, por exemplo, uma expressão regular s de comprimento m e alguns algoritmos conhecidos, todos não lineares, incluem os de Thompson [Tho68], de complexidade $O(mn)$, e o de Myers [Mye92], de complexidade $O(mn/\log n)$.

Outras generalizações admitem números limitados de erros na ocorrência do padrão s como fator de t, como veremos na Seção 5.2. Também nestes casos os algoritmos conhecidos não são lineares.

3.2 Fator comum mais longo

Outro problema de interesse é o problema da identificação de um fator comum a duas seqüências s e t que tenha comprimento máximo. A solução deste problema é necessária no processo de montagem de seqüências de DNA e também nas suas análises. Outros interesses incluem a identificação de casos de plágio entre trabalhos de alunos, ou estudos de similaridades de páginas na internet.

No Algoritmo 3.3 vemos uma solução ingênua para o problema. Neste caso, a complexidade do algoritmo é $O(m^3n)$ onde $m = |s|$ e $n = |t|$, o que é ainda mais insatisfatória que a do Algoritmo 3.1, cuja complexidade é $O(mn)$. De fato, cada busca à linha 5 é feita em tempo $O(mn)$ e um total

Algoritmo 3.3 Algoritmo $O(m^3n)$ de busca de fator comum mais longo

F C I ˆ (s, t)

1 ▹ devolve o mais longo fator comum às palavras s e t
2 $f \leftarrow 1$
3 **para** i **de** 1 **até** $|s|$ **faça**
4   **para** j **de** i **até** $|s|$ **faça**
5   **se** B I ˆ D P ˜ $(s[i\,..\,j], t)$
            e $|s[i\,..\,j]| > |f|$ **então**
6             $f \leftarrow s[i\,..\,j]$
7 **devolva** f

de $O(n^2)$ de tais buscas são realizadas devido às iterações às linhas 3 e 4. A análise para mostrar que não se pode fazer melhor que isto neste caso é ligeiramente mais complicada, mas também é simples. Considerando-se apenas os casos em que $i \leq m/3$ e $j \geq m/3$, temos que $m/3$ iterações[1] do laço à linha 3, cada uma delas com $m/3$ iterações do laço à linha 4, cada uma delas chamando B I ˆ D P ˜ $(s[i\,..\,j], t)$ com tempo de execução ao menos $mn/3$, já que $|s[i\,..\,j]| \geq m/3$ nestes casos. Isto já garante um tempo $m^3n/27$, que é $O(m^3n)$.

É imediato verificar que algumas melhorias podem ser introduzidas ao Algoritmo 3.3 de forma a reduzir sua complexidade computacional. Uma primeira mais imediata é observar que a chamada de B I ˆ - D P ˜ , de complexidade $O(mn)$, à linha 5 pode ser substituída pela

[1]Por simplicidade de notação, suporemos m divisível por 3.

chamada B D P ˜ $(s[i\,..\,j], t)$, de complexidade $O(m + n)$. Somente esta melhoria já reduz a complexidade total para $O(m^2(m + n))$. Se observarmos que cada uma das $O(m^2)$ chamadas calcula a árvore dos sufixos de t à linha 4 do Algoritmo 3.2, certamente podemos fatorar esta computação de forma a melhorar ligeiramente a complexidade para $O(m^3 + n)$. Este é o Algoritmo 3.4. Não é muito difícil observar que, fixado i à iteração da

Algoritmo 3.4 Algoritmo $O(m^3 + n)$ de busca de fator comum mais longo

F C C´ (s, t)
1 ▹ devolve o mais longo fator comum às palavras s e t
2 $f \leftarrow 1$
3 $\mathbf{T} \leftarrow$ Á D S (t)
4 **para** i **de** 1 **até** $|s|$ **faça**
5   **para** j **de** i **até** $|s|$ **faça**
6         $(h, u, x, v) \leftarrow$ S $(s[i\,..\,j], 1, \mathbf{T})$
7   **se** $h = s[i\,..\,j]$ **e** $|h| > |f|$ **então**
8             $f \leftarrow s[i\,..\,j]$
9 **devolva** f

linha 4, as sucessivas iterações de j à linha 5 levarão à tentativa de soletrar em $\mathbf{T}$ os sucessivos prefixos $s[i\,..\,i], s[i\,..\,i+1], \ldots, s[i\,..\,|s|]$ do mesmo sufixo $s[i\,..\,|s|]$. Pode-se assim eliminar as iterações de j à linha 5 e a condição $h = s[i\,..\,j]$ ao teste da linha 7. Resulta o Algoritmo 3.5, de complexidade computacional $O(m^2 + n)$.

Algoritmo 3.5 Algoritmo $O(m^2 + n)$ de busca de fator comum mais longo

F C Q (s, t)
1 ▹ devolve o mais longo fator comum às palavras s e t
2 $f \leftarrow 1$
3 $\mathbf{T} \leftarrow$ Á D S (t)
4 **para** i **de** 1 **até** $|s|$ **faça**
5     $(h, u, x, v) \leftarrow$ S $(s[i\,..\,|s|], 1, \mathbf{T})$
6   **se** $|h| > |f|$ **então**
7         $f \leftarrow h$
8 **devolva** f

Observe-se que as sucessivas tentativas de soletrar os sufixos

$$s[1\,..\,m], s[2\,..\,m], \ldots, s[m\,..\,m]$$

à linha 5 levam à parcela quadrática da complexidade $O(m^2 + n)$ do Algoritmo 3.5. Esta situação é bastante semelhante às sucessivas chamadas S $(w[i\,..\,n], 1, \mathbf{T})$ feitas à linha 9 do Algoritmo 2.2 quando A - P é chamada por Á D S no Algoritmo 2.3. Como naquele caso, podemos eliminar este "gargalo" e melhorar o Algoritmo 3.5 de forma a dar-lhe uma complexidade computacional linear através do uso das ligações de sufixos. No caso do Algoritmo 2.7, esta melhoria só é possível devido ao Lema 2.9. Assim temos o Lema 3.1, que nos mostra como aproveitar o cálculo da cabeça de $s[i\,..\,m]$ em $\mathbf{T}$ para o cálculo da cabeça de $s[i+1\,..\,m]$ em $\mathbf{T}$.

Lema 3.1 *Seja W um conjunto de palavras e* $\mathbf{T}$ *sua árvore de sufixos. Seja w uma palavra não vazia tal que sua cabeça h em* $\mathbf{T}$ *seja não vazia. Então $A^{-1}h$ é prefixo da cabeça de $A^{-1}w$ em* $\mathbf{T}$. *Ademais, se este prefixo é próprio e se o lugar de h em* $\mathbf{T}$ *está no meio de uma aresta, temos que $A^{-1}h$ é vértice de* $\mathbf{T}$.

Prova. A demonstração é bastante semelhante à do Lema 2.9. $\mathbf{T}$ é a A^+-árvore de reconhecimento de $\mathrm{Suf}(W)$. Do item 5 do Teorema 2.4, temos que a cabeça de uma palavra x em $\mathbf{T}$ é o seu mais longo prefixo que está em $\mathrm{Pref}(\mathrm{Suf}(W)) = \mathrm{Fat}(W)$. Seja $w' \in \mathrm{Suf}(W)$ tal que $h \in \mathrm{Pref}(w')$. Como $A^{-1}h$ é prefixo comum de $A^{-1}w$ e de $A^{-1}w' \in A^{-1}W \subseteq \mathrm{Suf}(W)$, segue que $A^{-1}h$ é prefixo da cabeça de $A^{-1}w$ em $\mathbf{T}$.

Suponha que este prefixo seja próprio, e que o lugar de h em $\mathbf{T}$ está na aresta (u, v). Primeiro, podemos escolher a a primeira letra de $h^{-1}w$. Assim temos que $A^{-1}ha$ é prefixo da cabeça de $A^{-1}w$ em $\mathbf{T}$, e portanto é prefixo de $\mathrm{Suf}(W)$. Segundo, seja b a primeira letra de $h^{-1}v$. Assim, $hb \in \mathrm{Pref}(v) \subseteq \mathrm{Pref}(\mathrm{Suf}(W))$ e $A^{-1}hb \in \mathrm{Pref}(A^{-1}v) \subseteq \mathrm{Pref}(A^{-1}\mathrm{Suf}(W)) \subseteq \mathrm{Pref}(\mathrm{Suf}(W))$. Terceiro, temos que $a \neq b$ pois do contrário teríamos que $ha \in \mathrm{Pref}(w) \cap \mathrm{Pref}(\mathrm{Suf}(W))$, que seria uma contradição com a definição de h. Finalmente, isto prova que $A^{-1}h$ é bifurcação de $\mathrm{Suf}(W)$, e portanto, é vértice de $\mathbf{T}$. ∎

Assim temos a solução final para o problema, através do Algoritmo 3.6, de complexidade computacional linear $O(m+n)$, como pode-se demonstrar de forma idêntica à do Algoritmo 2.7.

Algoritmo 3.6 Algoritmo $O(m + n)$ de busca de fator comum mais longo

```
F    C    (s, t)
▹ devolve o mais longo fator comum às palavras s e t
T ← Á      D   S      (t)
(h, u, x, v) ← S      (s, 1, T)
t ← h⁻¹s
f ← h
m ← |s|
para i de 2 até m faça
        ▹ ht = s[i − 1 . . m] e (u, x) é o lugar de h em T
        ▹ calcula cabeça h de s[i . . m] em T
        se x = 1 então
                se u = 1 então
                        (h, u, x, v) ← S      (A⁻¹t, 1, T)
                senão   (h, u, x, v) ← S      (t, S      (u), T)
        senão   se u = 1 então
                        (h, u, x, v) ← R S      (A⁻¹x, 1, T)
                senão   (h, u, x, v) ← R S      (x, S      (u), T)
                se x = 1 então
                        (h, u, x, v) ← S      (t, u, T)
        t ← h⁻¹s[i . . m]
        se |h| > |f| então
                f ← h
devolva f
```

Vale observar que o Algoritmo 3.6 revela a aplicabilidade das ligações de sufixos mesmo após a construção final da árvore dos sufixos. Este comportamento se repete em muitos dos algoritmos que fazem uso da árvore dos sufixos.

Outra nota não menos importante sobre o Algoritmo 3.6 é que a solução apresentada descreve um tipo de procedimento utilizado em vários outros procedimentos envolvendo árvores de sufixos. Definamos as *estatísticas de emparelhamento* de s em relação a t como sendo um vetor e de estatísticas em que em cada posição i guarda o comprimento do mais longo fator que ocorre na posição i de s e em qualquer posição em t. Ocorre que uma pequena alteração no Algoritmo 3.6 pode ser usada para computar as estatísticas de emparelhamento e. De fato, $e[i]$ é o comprimento da cabeça de $s[i\,..\,|s|]$ em $\mathbf{T}$, e podemos computar $e[1] = |h|$ logo após à linha 5 e os demais valores $e[i] = |h|$ logo antes da linha 20.

Por outro lado, o problema do mais longo fator comum admite uma solução envolvendo árvore de sufixos e que seja bem mais simples, como é descrito no Algoritmo 3.7. Na linha 4 temos uma busca em profundidade na árvore para calcular, para cada vértice v, quais das palavras s e t admitem sufixo associado a algum vértice final na sub-árvore de v. À linha 11 escolhemos o rótulo do vértice v com maior profundidade de palavra que seja fator de s e de t. Por outro lado, vale mencionar que as estatísticas de

Algoritmo 3.7 Algoritmo alternativo de busca de fator comum mais longo

F C (s, t)
1 ▹ devolve o mais longo fator comum às palavras s e t
2 $\mathbf{T} \leftarrow$ Á D S $(\{s, t\})$
3 $f \leftarrow 1$
4 **para** cada vértice v de $\mathbf{T}$ **faça**
5   ▹ verifica se a sub-árvore de v possui sufixos associados a s e t
6   **se** v é vértice final **então**
7   verifica se o(s) sufixo(s) associado(s) é de s e/ou de t
8   **para** cada filho u **faça**
9   verifica se a sub-árvore de u possui sufixos de s e de t
10   **se** a sub-árvore de v possui sufixos associados a s e t
      e C $(v) > |f|$ **então**
11         $f \leftarrow v$
12 **devolva** f

emparelhamento tornam-se praticamente a única alternativa em circunstâncias onde há necessidade de uma implementação envolvendo árvore dos sufixos e que faça uso mais moderado de memória. De fato, na primeira solução vista no Algoritmo 3.6, foi construída a árvore de sufixos de $\{t\}$, enquanto que na solução do Algoritmo 3.7 foi construída a árvore dos sufixos de $\{s, t\}$.

Capítulo 4

Menor Ancestral Comum em Árvores

Neste capítulo estudamos um algoritmo inacreditavelmente eficiente para encontrar o menor ancestral comum de dois vértices dados numa árvore. Além de bonito este algoritmo é útil também em várias aplicações. No meio do caminho resolveremos outros problemas correlatos, como, por exemplo, o problema de encontrar o elemento mínimo num intervalo dado de um vetor. Os melhores algoritmos necessitarão de um pré-processamento de tempo linear , após o que qualquer instância do problema poderá ser respondida em tempo constante (e portanto, independente do tamanho das estruturas envolvidas).

4.1 O problema e seu ambiente

Dado um vértice u de uma árvore existe um único caminho de u até a raiz r da árvore. Qualquer vértice neste caminho será dito um *ancestral* de u. Dados dois vértices, u e v numa árvore, eles sempre tem ancestrais comuns, a raiz é um exemplo de ancestral comum. Denominamos de *menor ancestral comum* de u e v o ancestral comum de u e de v que seja a mais distante da raiz. (O adjetivo menor refere-se ao ancestral comum "mais jovem" de u e de v). Denotaremos por $\mathrm{M}\quad(u,v)$ o menor ancestral comum de u e v. Por exemplo, na Figura 4.1, o menor ancestral comum dos vértices w e y é o vértice u, aquele de s e z é p e aquele de de o e z é m, isto é,

$$\mathrm{M}\quad(w,y)=u,\quad \mathrm{M}\quad(s,z)=p,\quad \mathrm{M}\quad(o,z)=m.$$

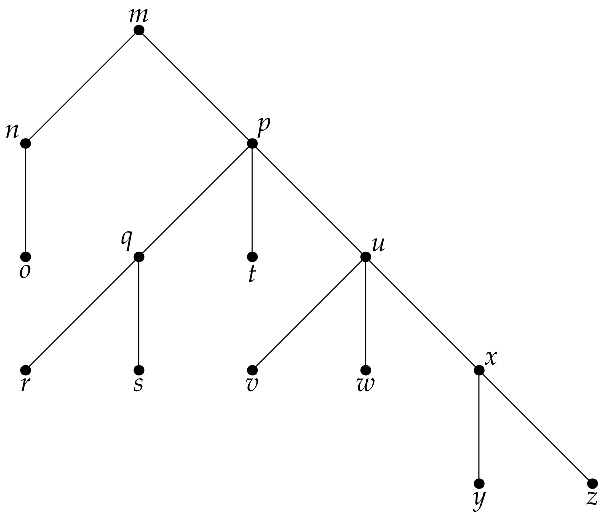


Figura 4.1: Árvore **T** para ilustrar o problema do menor ancestral comum

Antes de apresentarmos o algoritmo necessitamos fixar um modelo para a medida da sua complexidade. Os algoritmos a serem estudados neste capítulo terão sempre duas partes bem distintas. Assim, a sua complexidade será expressa em termos de um par ordenado de complexidades, como por exemplo em $\langle O(n), O(1)\rangle$. A primeira complexidade, $O(n)$ no caso, refere-se ao tempo usado por um algoritmo de pré-processamento. A segunda componente, $O(1)$ no caso, refere-se ao tempo usado para responder uma instância do problema, dado que o pré-processamento foi feito (e suas estruturas de dados estão disponíveis).

Desta forma, uma solução $\langle O(n), O(1)\rangle$ do problema de MAC implica que a árvore de n vértices sofre um pré-processamento linear. A partir deste pré-processamento qualquer instância do problema pode ser resolvida em tempo constante, independente do tamanho da árvore.

Vale a pena o leitor se deter por um momento para considerar a natureza desta complexidade. Intuitivamente a complexidade acima corresponde a um algoritmo de pré-processamento que demora um tempo constante em cada vértice da árvore dada. Nesta fase o algoritmo furiosamente faz anotações que poderão ser consultadas mais adiante. O pré-processamento produzirá tabelas de tamanho $O(n)$ no caso. Feito o pré-processamento qualquer instância do problema pode ser respondida em tempo constante, qualquer que tenha sido o tamanho da árvore envolvida.

Este modelo é particularmente útil quando precisamos responder várias instâncias do problema para uma mesma árvore. Neste caso a economia de tempo pode ser muito grande. Mas, mesmo no caso de uma única instância do problema a ser respondida para uma árvore dada, o algoritmo apresentado é ótimo, já que não podemos prescindir de examinar a árvore toda para responder a pergunta em geral.

Poucos problemas úteis apresentam soluções que possam ser respondidas em tempo constante, mesmo após um pré-processamento linear. Quando esta situação ocorre os algoritmos são particularmente sensíveis para a organização das tabelas intermediárias que podem depender de códigos e convenções engenhosas. Dada a natureza ótima do resultado, porém, é essencial compreender com profundidade os algoritmos envolvidos. E é isto que faremos neste capítulo com relação ao problema apresentado.

4.2 Representação de árvores

Seja **T** uma árvore sobre um conjunto V de vértices. A raiz de **T** será representada por R (**T**) e os demais dados da árvore serão representadas por uma tripla de funções:

$$\mathrm{P}\ (), \mathrm{P}\ \ (), \mathrm{I}\ \ \tilde{}\ () : V \to V \cup \quad .$$

Para cada vértice v, P (v) será o pai de v, a menos da raiz que obedecerá

$$\mathrm{P}\ (\mathrm{R}\ (\mathbf{T})) = \quad .$$

Os filhos de cada vértice serão enumerados numa ordem arbitrária mas fixa. Se o vértice v tiver filhos então um deles será designado de *primogênito* e será denotado de P (v). Os vértices v sem filhos, isto é as folhas de **T**, terão P (v) = . Cada vértice w terá um *irmão*, representado por I ˜ (w), que será ou será um outro vértice w', tal que P (w) = P (w').

Para um vértice v com $k \geq 0$ filhos a seqüência de $k + 1$ termos

$$\mathrm{P}\ \ (v), \mathrm{I}\ \ \tilde{}\ (\mathrm{P}\ \ (v)), \mathrm{I}\ \ \tilde{}\ (\mathrm{I}\ \ \tilde{}\ (\mathrm{P}\ \ (v))), \cdots$$

começará com exatamente k termos vértices, cada filho de v comparecendo exatamente uma vez, e a seqüência sempre terminará com .

Ilustramos a nossa representação de árvores na Tabela 4.1 que contém os valores das funções P (), P () e I ˜ () para a árvore **T** da Figura 4.1. Os filhos de cada vértice de **T** foram enumerados da esquerda para a direita, partindo-se do desenho daquela figura. Além dos dados da Tabela temos, também, R (**T**) = m.

v	m	n	o	p	q	r	s	t
P (**v**)		m	n	m	p	q	q	p
P (**v**)	n	o		q	r			
I ˜ (**v**)		p			t	s		u

v	u	v	w	x	y	z
P (**v**)	p	u	u	u	x	x
P (**v**)	v			y		
I ˜ (**v**)		w	x		z	

Tabela 4.1: A representação da árvore **T** da Figura 4.1

4.3 Mac em tempos $\langle O(1), O(n)\rangle$ e $\langle O(n^2), O(1)\rangle$

Apresentamos agora dois algoritmos ingênuos para a solução do nosso problema. A primeira solução terá complexidade $\langle O(1), O(n)\rangle$, isto é não temos pré-processamento no caso e cada consulta leva tempo proporcional ao tamanho da árvore.

Para computar M (u, v) o Algoritmo 4.1 percorre os ancestrais de u e de v e descobre o ancestral comum de maior distância da raiz. Os vetores A e B guardam os ancestrais de u e v, respectivamente. O Algoritmo 4.2 é uma função auxiliar, A (v), que coloca num vetor os ancestrais de v, começando com v e terminando com a raiz da árvore.

Algoritmo 4.1 M (u, v) em tempo $\langle O(1), O(n)\rangle$ para um grafo **T** de n vértices

M I ˆ (u, v)
▹ A e B são vetores de vértices de **T**
$(i, A) \leftarrow$ A (u)
$(j, B) \leftarrow$ A (v)
enquanto $i \geq 1$ **e** $j \geq 1$ **e** $A[i] = B[j]$ **faça**
    $i \leftarrow i - 1$
    $j \leftarrow j - 1$
devolva $A[i+1]$ ▹ $A[i+1]$ é o M (u, v)

Este algoritmo pode ser melhorado, sem melhorar, porém, a sua complexidade, se tivermos à disposição as profundidades[1] dos vértices da árvore **T**. Neste caso, subimos primeiro o vértice de maior profundidade, indo sempre na direção da raiz, até encontrarmos o primeiro vértice comum nos vetores A e B. Deixamos os detalhes ao cargo do leitor.

Algoritmo 4.2 Cálculo dos ancestrais

```
A          (v)
  ▷ coloca os ancestrais de v no vetor A[1 . . i]
  i ← 1
  u ← v
  enquanto P   (u) ≠        faça
        A[i] ← u
        i ← i + 1
        u ← P   (u)
  A[i] ← u                  ▷ A[i] é a raiz de T
  devolva (i, A)            ▷ Retornamos i e A
```

O algoritmo M I ˆ pode ser usado para a obtenção de uma solução $\langle O(n^3), O(1)\rangle$ para o Problema do Menor Ancestral Comum. Isto mostra, pelo menos, que uma solução com consulta em tempo constante é possível. De fato, basta calcular, na fase de pré-processamento, uma matriz $V \times V$, cujo elemento na linha u e coluna v conterá M (u, v). O pré-processamento terá complexidade $O(n^3)$ ($O(n^2)$ processamentos de M I ˆ , um para cada par não-ordenado $\{u, v\}$ de vértices de **T**). Feito o pré-processamento, o cálculo de M (u, v) leva tempo constante: basta fazer uma consulta à matriz pré-calculada.

Deixamos ao cargo do leitor a melhora do esquema acima para a obtenção de uma solução $\langle O(n^2), O(1)\rangle$. Sugerimos, mais uma vez, usar as profundidades dos vértices e procurar evitar a repetição de cálculos. Para tanto, pode-se valer do fato de que em muitos casos M (u, v) = M (P $(u), v)$ e/ou M (u, v) = M $(u,$ P $(v))$.

Isto posto, a nossa tarefa, daqui em diante, é reduzir o tempo do pré-processamento de $O(n^2)$ para $O(n)$, sem comprometer a consulta em tempo constante.

[1] A profundidade de um vértice numa árvore foi definida no Capítulo 1, ela será calculada no Algoritmo 4.3.

4.4 Busca em profundidade de árvores

Dada uma árvore **T** com n vértices, calculamos nesta seção um vetor com $m = 2n - 1$ vértices, que descreve, passo a passo, a busca em profundidade de **T**. Chamaremos este vetor de B , estando o grafo **T** subentendido. Calcularemos, também, o vetor P , cujos elementos serão, respectivamente, as profundidades dos vértices em B , lembrando que a profundidade de um vértice v de **T** é a distância de v à raiz de **T**. Os vetores B e P serão indexados de 1 a m. O índice dos elementos serve também como um parâmetro de "cronometragem" do algoritmo da busca em profundidade: se B $[i] = u$ então dizemos que a busca visitou o vértice u no instante i. Registraremos em D $[u]$ o instante da primeira visita ao vértice u e em A $[u]$ o instante da última visita a u. Estes instantes são, respectivamente, a *descoberta* e o *abandono* de u. Estes cálculos serão feitos no Algoritmo 4.3 em tempo e espaço $O(n)$. Eles permitirão reduzir, na seção seguinte, o problema do Menor Ancestral Comum a um outro problema, aparentemente mais simples: o problema do Mínimo de um Vetor Contínuo, a ser abordado na próxima seção.

A Busca em Profundidade de um grafo é um algoritmo clássico nas Teorias dos Grafos e da Computação. O seu estudo detalhado está fora do escopo deste livro; suporemos que o leitor está familiarizado com os conceitos envolvidos. Uma excelente referência é o Capítulo 23, "Elementary Graph Algorithms" do livro de Cormen, Leiserson e Rivest [CLR90].

A nossa busca em profundidade inicia-se no instante 1 na raiz da árvore **T**. Em cada instante a busca está visitando um vértice; se no instante i o vértice visitado é v então teremos

$$\mathrm{B}\qquad [i] = v.$$

Escolhe-se o próximo vértice a ser visitado de acordo com os critérios abaixo:

1. Se v tiver algum filho, u, ainda não visitado então visita-se u a seguir; isto corresponde a percorrer a aresta $\{u, v\}$ na direção de afastar-se da raiz, i.e. descemos a aresta $\{u, v\}$.

2. Se todos os filhos de v já foram visitados e v não é a raiz da árvore então (re)visita-se a seguir P $(v) = u$; isto corresponde a percorrer a aresta $\{u, v\}$ na direção de aproximar-se da raiz, i.e. subimos a aresta $\{u, v\}$.

Algoritmo 4.3 A B E P da árvore **T**, de n vértices, devolve, em tempo $O(n)$, os vetores B e P de $m = 2n - 1$ elementos e os vetores D e A de n elementos

```
B E P (T)
▹ D , A e A são vetores indexados pelos vértices de T
▹ B e P são indexados pela "cronometragem" do algoritmo
para u ∈ V faça
        A[u] ← P (u)                    ▹ A[u]: filho não descoberto de u
v ← R (T)
i ← 1
B [i] ← v
P [i] ← 0
D [v] ← 1
enquanto P (v) ≠ ou A[v] ≠ faça
        i ← i + 1
        se A[v] ≠ então
                w ← A[v]                ▹ Descemos
                P [i] ← P [i − 1] + 1
                D [w] ← i               ▹ Registramos a descoberta de w
                A[v] ← I ˜ (A[v])       ▹ Novo filho não descoberto de v
        senão   w ← P (v)               ▹ Subimos
                P [i] ← P [i − 1] − 1
                A [v] ← i − 1           ▹ Registramos o abandono de v
        v ← w
        B [i] ← v
m ← i
A [v] ← m
devolva (m, B , P , D , A )
```

3. Se todos os filhos de v já foram visitados e v é a raiz de **T**, isto é, P (v) = , então a busca terminou. Guardamos o instante do término em m.

Não é difícil ver que a busca em profundidade da árvore **T** percorrerá cada aresta da árvore duas vezes, uma vez em cada direção: afastando-se e aproximando-se da raiz. Como a árvore tem $n-1$ arestas, a busca termina no instante $m = 2n-1$, na raiz de **T**. Esta observação garante que a complexidade do Algoritmo 4.3, B E P , é $O(n) = O(m)$.

Os vetores D e A são indexados pelos vértices de **T**. O valor de D $[u]$ é o instante da primeira visita a u. O de A $[u]$ é o instante da última visita a u. Veremos adiante que os intervalos[2] [D $[u]$ . . A $[u]$] impõem uma estrutura de parênteses no histórico da busca em profundidade.

Acompanhando o cálculo de B $[i]$ registramos também, em P $[i]$ a profundidade do vértice visitado no instante i. Note que valores consecutivos deste vetor variam de ± 1, já que em cada instante da busca ou subimos ou descemos uma aresta. Esta propriedade será fortemente explorada a seguir. Note, também, que as computações podem ser usadas para determinar, em tempo $O(1)$, a profundidade dos vértices. De fato, a profundidade do vértice u é dada por

P [D $[u]$] ou P [A $[u]$].

Exemplificamos os conceitos e procedimentos acima com a árvore da Figura 4.1. Numa busca em profundidade, usando a representação da Tabela 4.1, os vértices de **T** são descobertos na ordem abaixo, que (não por acaso) é a lista dos nomes dos vértices em ordem alfabética:

$$m, n, o, p, q, r, s, t, u, v, w, x, y, z.$$

Os vetores B e P do grafo **T** da Figura 4.1, calculados pelo Algoritmo 4.3, são dados na Tabela 4.2. Os vetores D e A são apresentados na Tabela 4.3.

Uma propriedade fundamental da busca em profundidade é dada pelo Teorema dos Parênteses [CLR90, página 480] que no nosso contexto, já especializado para a busca em profundidade de árvores, exploradas a partir da sua raiz, pode ser assim enunciado:

[2]Todos os intervalos que consideraremos neste capítulo serão fechados e como de costume serão denotados por $[a \mathinner{..} b]$

i	1	2	3	4	5	6	7	8
B [i]	m	n	o	n	m	p	q	r
P [i]	0	1	2	1	0	1	2	3

i	9	10	11	12	13	14	15	16
B [i]	q	s	q	p	t	p	u	v
P [i]	2	3	2	1	2	1	2	3

i	17	18	19	20	21	22	23	24	25	26	27
B [i]	u	w	u	x	y	x	z	x	u	p	m
P [i]	2	3	2	3	4	3	4	3	2	1	0

Tabela 4.2: Histórico e profundidades dos vértices visitados na busca em profundidade da árvore **T**.

v	m	n	o	p	q	r	s	t	u	v	w	x	y	z
D [**v**]	1	2	3	6	7	8	10	13	15	16	18	20	21	23
A [**v**]	27	4	3	26	11	8	10	13	25	16	18	24	21	23

Tabela 4.3: A estrutura parentética da busca em profundidade da árvore **T**: instantes de descoberta e de abandono dos vértices.

Teorema 4.1 *Na busca em profundidade de uma árvore* **T**, *iniciada na sua raiz, para um vértice v seja I_v o intervalo de números inteiros dado por* $I_v = [\mathrm{D}\quad[v]\,..\,\mathrm{A}\quad[v]]$. *Para quaisquer vértices distintos u e v exatamente uma das afirmações abaixo é verdadeira:*

- *os intervalos I_u e I_v são disjuntos, isto é ou* $\mathrm{D}\quad[u] \leq \mathrm{A}\quad[u] < \mathrm{D}\quad[v] \leq \mathrm{A}\quad[v]$ *ou* $\mathrm{D}\quad[v] \leq \mathrm{A}\quad[v] < \mathrm{D}\quad[u] \leq \mathrm{A}\quad[u]$; *neste caso u e v são independentes na relação de ancestralidade de* **T**.
- *o intervalo I_u está contido no intervalo I_v, isto é* $\mathrm{D}\quad[v] < \mathrm{D}\quad[u] \leq \mathrm{A}\quad[u] < \mathrm{A}\quad[v]$; *neste caso v é ancestral de u em* **T**.
- *o intervalo I_v está contido no intervalo I_u, isto é* $\mathrm{D}\quad[u] < \mathrm{D}\quad[v] \leq \mathrm{A}\quad[v] < \mathrm{A}\quad[u]$; *neste caso u é ancestral de v em* **T**.

Para exemplificar o Teorema 4.1, vamos nos concentrar no exemplo

$$\mathrm{M}\quad(s, z) = p.$$

Note que s e z são incomparáveis na ordem parcial de ancestralidade, isto é, nenhum dos dois é ancestral do outro. De fato, usando a notação I_v do Teorema, os intervalos

$$I_s = [10\,..\,10] \quad \text{e} \quad I_z = [23\,..\,23]$$

são disjuntos. O vértice p é (o menor) ancestral comum de s e de z. De acordo, ambos os intervalos acima estão contidos em $I_p = [6\,..\,26]$. Note que este é o menor intervalo da forma I_x que contém tanto I_s quanto I_z.

Os intervalos do vetor B , da forma

$$\mathrm{B}\qquad[\mathrm{D}\quad[x]\,..\,\mathrm{A}\quad[x]] = \mathrm{B}\qquad[I_x]$$

tem uma estrutura muito bem determinada que desejamos ilustrar com o exemplo de $x = p$. O leitor verificará que

$$\mathrm{B}\qquad[I_p] = \mathrm{B}\qquad[6\,..\,26] = (p, \mathrm{B}\qquad[I_q], p, \mathrm{B}\qquad[I_t], p, \mathrm{B}\qquad[I_u], p).$$

Note que q, t e u são os filhos de p. Note também que I_s está contido em I_q e que I_z está contido em I_u. Para uso futuro, note também que p é o vértice de maior profundidade no intervalo

$$[\mathrm{D}\quad[s]\,..\,\mathrm{D}\quad[z]] = [10\,..\,23].$$

Usando o Teorema acima pode-se demonstrar o resultado seguinte, fundamental para o entendimento do resultado principal do Capítulo. Não faremos a sua demonstração, pois ela depende de uma análise cuidadosa da Busca em Profundidade e para a qual não introduzimos uma linguagem suficientemente rica neste livro. Acreditamos, porém, que o exemplo acima da estrutura da exploração do vértice p descreve as idéias principais que intervém na demonstração do Teorema 4.2.

Teorema 4.2 *Sejam u e v vértices da árvore* $\mathbf{T}$*, tais que* $\mathrm{D}\ \ [u] < \mathrm{D}\ \ [v]$*. Então, o menor ancestral comum de u e v em* $\mathbf{T}$ *é o (único) vértice de profundidade mínima no intervalo* $\mathrm{B}\ \ [\mathrm{D}\ \ [u]\,..\,\mathrm{D}\ \ [v]]$.

A análise cuidadosa da complexidade do Algoritmo 4.3, da Busca em Profundidade, também é parte do estudo da Busca em Profundidade e remetemos o leitor para o livro já citado de Cormen, Leiserson e Rivest. A análise é baseada no fato de que $m = 2n - 1$ e que o algoritmo processa cada aresta de $\mathbf{T}$ duas vezes: uma vez descendo a aresta e, mais para a frente, subindo a mesma aresta. Note que se $a = (u, v)$ for uma aresta de $\mathbf{T}$, com u o seu extremo mais próximo da raiz, então a descida de a se dá exatamente no instante $\mathrm{D}\ \ [v]$, isto é,

$$\mathrm{B}\ \ [\mathrm{D}\ \ [v] - 1] = u \quad \text{e} \quad \mathrm{B}\ \ [\mathrm{D}\ \ [v]] = v.$$

A subida desta mesma aresta se dá no instante $\mathrm{A}\ \ [v]$, isto é,

$$\mathrm{B}\ \ [\mathrm{A}\ \ [v]] = v \quad \text{e} \quad \mathrm{B}\ \ [\mathrm{A}\ \ [v] + 1] = u.$$

Segue o resultado sobre a complexidade linear do Algoritmo 4.3.

4.5 Mínimo de um vetor contínuo

4.5.1 Um novo problema

Apresentamos um problema, relacionado ao cálculo de $\mathrm{M}\ \ (u, v)$ através da redução acima, e cuja solução eficiente vai levar à solução procurada para o problema original.

Seja $V[1\,..\,m]$ um vetor de m números inteiros. Dizemos que o vetor V é *contínuo* se a diferença entre elementos consecutivos do vetor é ± 1, isto é, para cada i, $1 < i \leq m$, $|V[i] - V[i-1]| = 1$.

Vetores contínuos de m elementos estão em correspondência biunívoca com fatores de comprimento $m-1$ de palavras balanceadas de parênteses, isto segue das propriedades da linguagem de Dyck, estudadas na Teoria de Linguagens Livres de Contexto, ou das propriedades dos números de Catalan, estudadas em Combinatória. Estas ligações são irrelevantes para o nosso problema (são mesmo?), por isto não entraremos mais a fundo nestas questões.

O nosso novo problema recebe, como dado, um vetor contínuo $V[1\,..\,m]$. Queremos fazer um pré-processamento em V para podermos responder, em tempo constante, a pergunta: dados índices i e j, $1 \leq i \leq j \leq m$, qual é a posição do elemento mínimo do intervalo $V[i\,..\,j]$? Denotaremos por $\mathrm{M}\quad(i,j)$ uma posição do mínimo procurado, evidentemente,

$$i \leq \mathrm{M}\quad(i,j) \leq j,$$

e a função M satisfaz a seguinte identidade

$$\mathrm{V}[\mathrm{M}\quad(i,j)] = \min\{\,\mathrm{V}[i], \mathrm{V}[i+1], \ldots, \mathrm{V}[j]\,\}.$$

O vetor de profundidades P $[i]$ da Tabela 4.2 é um exemplo de um vetor contínuo. Usaremos ele para exemplificar o novo problema: notamos que

$$\mathrm{M}\quad(18,21) = 2, \quad \mathrm{M}\quad(10,23) = 1, \quad \mathrm{M}\quad(3,23) = 0.$$

O leitor atento perceberá que estes exemplos correspondem aos exemplos apresentados na Seção 4.1. De fato, os vértices nos extremos dos intervalos dados e aqueles na posição dos mínimos encontrados são exatamente os vértices daqueles exemplos:

$$\mathrm{M}\quad(w,y) = u, \quad \mathrm{M}\quad(s,z) = p, \quad \mathrm{M}\quad(o,z) = m.$$

Lembrando-se do Teorema 4.2, a correspondência acima e a importância de uma solução eficiente do problema do Mínimo de um Vetor Contínuo ficam aparentes.

Mostraremos, a seguir, que existe um par de algoritmos de complexidade $\langle O(n), O(1)\rangle$ para resolver o Problema do Mínimo de um Vetor Contínuo. Este algoritmo, por sua vez, será usado na obtenção do resultado principal do Capítulo.

4.5.2 Mvc em tempo $\langle O(m \log m), O(1)\rangle$

Nesta seção apresentamos um par de algoritmos, P ´I ´ e M - I ´, especificados nos Algoritmos 4.6 e 4.7, respectivamente. Este par resolve o Problema do Mínimo de um Vetor Contínuo com complexidade $\langle O(m \log m), O(1)\rangle$ e constitui, portanto, uma melhora substancial sobre o algoritmo ingênuo para a solução do problema, que teria complexidade $\langle O(m^2), O(1)\rangle$, conforme o leitor poderá facilmente comprovar.

Os algoritmos desta seção serão chamados de *intermediários* já que eles situam-se entre os algoritmos ótimos, nosso objetivo, e os algoritmos ingênuos. Eles baseiam-se na seguinte propriedade fundamental. Sejam $[i \,..\, j]$ e $[i' \,..\, j']$ dois intervalos cuja união é o intervalo $[i \,..\, j']$. Mais precisamente, sejam i, i', j e j' tais que

$$i < i' \leq j < j' \quad \text{ou} \quad i \leq j < j + 1 = i' \leq j'.$$

No primeiro caso os dois intervalos se sobrepõem, no segundo eles são disjuntos. Em qualquer caso, para qualquer vetor V de números reais (não necessariamente um vetor contínuo)

$$\min V[i \,..\, j'] = \min\{\, \min V[i \,..\, j], \min V[i' \,..\, j']\,\}. \tag{4.1}$$

Visando explorar a propriedade acima iremos pré-calcular a posição dos mínimos em intervalos de V que tenham um número de elementos que são potências de dois, ou seja, 1, 2, 4, 8 ... elementos. Mais precisamente, para i, j e k[3] tais que $1 \leq i \leq i + 2^k - 1 = j \leq m$ colocaremos em PosMin$[i, k]$ o índice t, no intervalo fechado $[i \,..\, j]$, que satisfaz $V[t] = \min V[i \,..\, j]$. Ou seja, teremos a identidade:

$$\text{V}[\text{PosMin}[i, k]] = \min V[i \,..\, j].$$

Usando a propriedade 4.1, os valores pré-computados da matriz PosMin podem ser usados para o cálculo, em tempo constante, de M (i, j) da seguinte forma: sejam $k = \text{Log}[j - i + 1]$ e $i' = j - 2^k + 1$, então

$$\text{M}\quad(i, j) = \begin{cases} \text{PosMin}[i, k] & \text{se V}[\text{PosMin}[i, k]] \leq \text{V}[\text{PosMin}[i', k]], \\ \text{PosMin}[i', k] & \text{caso contrário.} \end{cases}$$

[3]Note que os três parâmetros i, j e k são dependentes. Quaisquer dois deles determina o terceiro.

Por sua vez, a própria computação dos valores de PosMin pode ser feita usando-se a propriedade 4.1. Basta procedermos em ordem crescente dos valores de k durante o cálculo dos PosMin$[i,k]$. Mais precisamente, colocamos PosMin$[i,0] = i$, para cada i, e

$$\mathrm{PosMin}[i,k] = \begin{cases} \mathrm{PosMin}[i,k-1] & \text{se } \mathrm{V}[p_i] \leq \mathrm{V}[p_{i'}], \\ \mathrm{PosMin}[i',k-1] & \text{caso contrário;} \end{cases}$$

onde $i' = i + 2^{k-1}$, $p_i = \mathrm{PosMin}[i,k-1]$ e $p_{i'} = \mathrm{PosMin}[i',k-1]$.

Exemplificamos a matriz PosMin com a Tabela 4.4 que corresponde ao nosso exemplo. Os números entre parênteses na Tabela são o extremo superior, j, dos respectivos intervalos envolvidos no cálculo de PosMin$[i,k]$.

Algoritmo 4.4 Cálculo de Logaritmos e de Potências

L EP ˆ (m)

```
k ← 0
Exp[k] ← 1
para i de 1 até m faça
        se i > Exp[k] então
                k ← k + 1
                Exp[k] ← 2 * Exp[k − 1]
        Log[i] ← k
devolva (Log, Exp)
```

Os algoritmos, daqui em diante, necessitarão do cálculo, em tempo constante, do logaritmo, na base 2, de inteiros entre 1 e m e do valor das potências de 2 que caem no intervalo $[1..m]$. Para conseguirmos isto faremos um pré-processamento de complexidade $O(m)$ guardando estes resultados nos vetores Log e Exp, que serão consultados em tempo constante durante os cálculos. Mais precisamente, temos:

$$\mathrm{Log}[i] = \lfloor \log_2 i \rfloor, \qquad \text{para } 1 \leq i \leq m,$$

$$\mathrm{Exp}[k] = 2^k, \qquad \text{para } 0 \leq k \leq \mathrm{Log}[m],$$

Os vetores Log e Exp são calculados no Algoritmo 4.4 e dispensam maiores comentários. Notamos apenas que a complexidade computacional deste algoritmo é $O(m)$.

i	1		2		3		4		5		6	
$k = 0$	(1)	1	(2)	2	(3)	3	(4)	4	(5)	5	(6)	6
$k = 1$	(2)	1	(3)	2	(4)	4	(5)	5	(6)	5	(7)	6
$k = 2$	(4)	1	(5)	5	(6)	5	(7)	5	(8)	5	(9)	6
$k = 3$	(8)	1	(9)	5	(10)	5	(11)	5	(12)	5	(13)	6
$k = 4$	(16)	1	(17)	5	(18)	5	(19)	5	(20)	5	(21)	6

i	7		8		9		10		11		12	
$k = 0$	(7)	7	(8)	8	(9)	9	(10)	10	(11)	11	(12)	12
$k = 1$	(8)	7	(9)	9	(10)	9	(11)	11	(12)	12	(13)	12
$k = 2$	(10)	7	(11)	9	(12)	12	(13)	12	(14)	12	(15)	12
$k = 3$	(14)	12	(15)	12	(16)	12	(17)	12	(18)	12	(19)	12
$k = 4$	(22)	12	(23)	12	(24)	12	(25)	12	(26)	12	(27)	27

i	13		14		15		16	
$k = 0$	(13)	13	(14)	14	(15)	15	(16)	16
$k = 1$	(14)	14	(15)	14	(16)	15	(17)	17
$k = 2$	(16)	14	(17)	14	(18)	15	(19)	17
$k = 3$	(20)	14	(21)	14	(22)	15	(23)	17

i	17		18		19		20	
$k = 0$	(17)	17	(18)	18	(19)	19	(20)	20
$k = 1$	(18)	17	(19)	19	(20)	19	(21)	20
$k = 2$	(20)	17	(21)	19	(22)	19	(23)	20
$k = 3$	(24)	17	(25)	19	(26)	26	(27)	27

i	21		22		23		24	
$k = 0$	(21)	21	(22)	22	(23)	23	(24)	24
$k = 1$	(22)	22	(23)	22	(24)	24	(25)	25
$k = 2$	(24)	22	(25)	25	(26)	26	(27)	27

i	25		26		27	
$k = 0$	(25)	25	(26)	26	(27)	27
$k = 1$	(26)	26	(27)	27		

Tabela 4.4: A matriz PosMin para o pré-processamento $O(m \log m)$

Algoritmo 4.5 Uma idéia chave num procedimento auxiliar

P L´ (i, j)

$k \leftarrow \mathrm{Log}[j - i]$
▹ É o valor de k que garante que o intervalo $[i\,..\,j]$ está coberto.
se $V[\mathrm{PosMin}[i, k]] \leq V[\mathrm{PosMin}[j - \mathrm{Exp}[k] + 1, k]]$ **então**
    devolva $\mathrm{PosMin}[i, k]$
senão **devolva** $\mathrm{PosMin}[j - \mathrm{Exp}[k] + 1, k]$

O procedimento auxiliar P L´ (i, j) calcula a posição do elemento mínimo no intervalo i a j, usando as posições pré-calculadas, na matriz PosMin, dos mínimos de intervalos de amplitude 2^k, para algum k. Este procedimento é a implementação da propriedade 4.1. Note que P L´ calcula o único valor de k que garante que os dois intervalos $[i\,..\,i + 2^k - 1]$ e $[j - 2^k + 1, j]$ cobrem o intervalo $[i, j]$. O leitor atento notará que M (i, j) = P L´ (i, j) e poderá perguntar porque usar dois símbolos para a mesma função? O motivo é que nos algoritmos da próxima seção esta igualdade desaparecerá.

Algoritmo 4.6 Pré-processamento em tempo $O(m \log m)$ para M

P ´I ´ (V)

$(\mathrm{Log}, \mathrm{Exp}) \leftarrow$ L EP ˆ (m)
para i **de** 1 **até** m **faça**
    $\mathrm{PosMin}[i, 0] \leftarrow i$
para k **de** 1 **até** $\mathrm{Log}[m]$ **faça**
    para i **de** 1 **até** $m - \mathrm{Exp}[k]$ **faça**
        $\mathrm{PosMin}[i, k] \leftarrow$ P L´ $(i, i + \mathrm{Exp}[k] - 1)$
devolva PosMin

Resta, apenas, examinarmos a complexidade dos Algoritmos 4.6 e 4.7. Está claro que M I ´ tem complexidade $O(1)$. Quanto ao algoritmo P ´I ´, o procedimento L EP ˆ tem complexidade $O(m)$ e, portanto, a complexidade de P ´I ´ é dominada pelo número de elementos da matriz PosMin. Como PosMin tem m linhas e cada linha tem no máximo $\log_2 m$ colunas, segue que a complexidade de P ´I ´ é $O(m \log m)$.

Algoritmo 4.7 Consulta em tempo constante para M

M I ´ (i, j)
1 **devolva** P L´ (i, j)

4.5.3 Mvc em tempo $\langle O(m), O(1)\rangle$

Apresentamos aqui um par de algoritmos, de complexidade $\langle O(m), O(1)\rangle$, para o cálculo de M (i, j). São os Algoritmos 4.14 e 4.15, que constituem o resultado principal da seção 4.5. Eles precisarão de vários procedimentos auxiliares, mas é importante manter em mente que estamos elaborando uma reorganização da solução apresentada na seção anterior. A reorganização passa por uma série de detalhes que tornam a exposição longa, mas os conceitos envolvidos são simples.

A idéia básica é esta: temos que diminuir o espaço ocupado pela tabela PosMin que usa $O(m \log m)$ posições. Uma forma de obter isto é particionar o vetor V em blocos de b elementos, com b proporcional a $\log m$, e construir a matriz PosMin apenas para auxiliar em consultas que começam e terminam em fronteiras entre blocos vizinhos, isto é, consultas que envolvem uniões de blocos inteiros. Naturalmente, com este PosMin restrito não poderemos responder muitas das consultas possíveis. Felizmente, estas respostas podem ser recuperadas através de uma técnica chamada dos "quatro russos"[4]. Esta técnica pode ser usada para levar em conta o que ocorre dentro dos blocos extremos envolvidos numa consulta.

A técnica dos "quatro russos" consiste, no caso, em pré-computar e estocar todos os M 's internos aos blocos. Veremos que isto pode ser feito em tempo e espaço $O(m)$ para blocos de $\log m$ elementos. Usaremos fortemente a condição da continuidade do vetor dado para obter a complexidade linear.

Tendo o pré-processamento dos resultados internos aos blocos, qualquer consulta pode ser respondida encontrando, em geral, o mínimo de três mínimos parciais:

- o mínimo interno a um bloco inicial,
- o mínimo de uniões de blocos inteiros e

[4]Veja uma explicação geral sobre a técnica dos "quatro russos" na seção 12.7 do livro de Gusfield [Gus97].

- o mínimo interno a um bloco final.

O caso que sobra é quando a consulta está inteiramente contida num só bloco: o pré-processamento fornece a resposta. Em qualquer caso, a consulta é respondida em tempo constante! Vejamos, agora, os detalhes.

No que segue, escolhemos um natural $b > 1$ que será o tamanho dos blocos. O valor de b será fixado no final da sub-seção em Log[m] mas preferimos manter a discussão independente deste valor. No nosso exemplo usaremos $b = 4$. A primeira providência é subdividir o vetor V em blocos de b elementos. Os intervalos de índices correspondentes aos primeiros g blocos serão, respectivamente,

$$[1\,..\,b], [b+1\,..\,2b], \ldots, [(g-1)b+1\,..\,gb].$$

O último bloco de V poderá ser incompleto, dependendo dos valores de m e b. O número de blocos completos que teremos será $\lfloor m/b \rfloor$ enquanto o número total de blocos será $g_m = \lceil m/b \rceil$.

O posicionamento de um índice i do vetor V agora será expressa por duas coordenadas: diremos que i está no g_i-ésimo bloco, com $1 \leq g_i \leq g_m$ e que dentro deste bloco i ocupa a posição p_i, $1 \leq p_i \leq b$. Evidentemente, existe uma bijeção entre os índices i e os pares (g_i, p_i). O valor do par (g_i, p_i) correspondente a i é dado pelo procedimento A (i), apresentado no Algoritmo 4.8. A função inversa, que não será utilizada, seria, para $g \geq 1$ e $1 \leq p \leq b$,

$$\text{D} \qquad (g, p) = (g-1) * b + p.$$

Nitidamente, o procedimento A tem complexidade $O(1)$. Ele usa um operador, que denominamos de *divisão inteira*, denotada por x **div** d, que associa a inteiros positivos x e d, $d \neq 0$, o par (q, r) de inteiros, onde q é o quociente da divisão inteira de x por d e r é o resto desta divisão. Temos: $q = \lfloor x/d \rfloor$, $0 \leq r < d$ e $x = qd + r$. No nosso exemplo o vetor V é o P e a sua partição em blocos de $b = 4$ elementos está feita na tabela 4.2.

A próxima observação é essencial para a obtenção da complexidade do nosso algoritmo. A posição do mínimo de um vetor contínuo não é alterada por uma translação do vetor. Ou seja, se $V[1\,..\,n]$ for um vetor contínuo então $V'[1\,..\,n]$, onde $V'[i] = V[i] + t$, para algum inteiro t, é contínuo também e, os mínimos de V e de V' ocorrem exatamente nas mesmas posições. Isto permite trabalharmos apenas com vetores "normais" onde diremos que um vetor contínuo é *normal*, se o seu primeiro elemento for 0.

Algoritmo 4.8 Cálculo do grupo e da posição do índice i

```
A (i)
(q, r) ← i div b
se r = 0 então
        devolva (q, b)
senão devolva (q + 1, r)
```

Podemos representar, biunivocamente, um vetor contínuo normalizado de n elementos pela seqüência das $n-1$ diferenças entre elementos sucessivos do vetor. Tais seqüências de diferenças serão denotadas por palavras de comprimento $n-1$ sobre o alfabeto $A = \{\mathtt{d}, \mathtt{s}\}$ de duas letras. Seja $N[1\,..\,n]$ um vetor contínuo normalizado. As correspondências entre as letras e as diferenças que usaremos são:

$$\mathtt{d} \text{ representa } N[i\,..\,i+1] \text{ se } N[i+1] - N[i] = -1 \text{ e}$$

$$\mathtt{s} \text{ representa } N[i\,..\,i+1] \text{ se } N[i+1] - N[i] = 1.$$

Necessitaremos de um vetor de duas posições para expressar esta correspondência, por isto, definimos:

$$\text{V ¸˜ } [\mathtt{d}] = -1 \quad \text{e} \quad \text{V ¸˜ } [\mathtt{s}] = 1.$$

A idéia por trás desta notação é que a letra d representa uma diminuição em N. Analogamente, a letra s representa um acréscimo em N.

Por exemplo, o vetor contínuo normalizado N de 6 elementos

$$N[1\,..\,6] = (0, 1, 0, -1, -2, -1) \tag{4.2}$$

será representado pela palavra v de comprimento 5

$$v = \mathtt{sddds}. \tag{4.3}$$

A seguir, definimos uma bijeção

$$\gamma : A^* \to \mathbb{N}_+$$

entre as palavras sobre o alfabeto $A = \{\mathtt{d}, \mathtt{s}\}$ e os números naturais não nulos. Postulamos que $\mathtt{d} < \mathtt{s}$ e associamos o natural $n > 0$ à n-ésima palavra de A^* na ordem militar, isto é, na ordem

$$\lambda, \mathtt{d}, \mathtt{s}, \mathtt{dd}, \mathtt{ds}, \mathtt{sd}, \mathtt{ss}, \mathtt{ddd}, \mathtt{dds}, \ldots.$$

A bijeção γ é uma enumeração das palavras de A^* e é amplamente utilizada na Teoria da Computação. A partir da representação binária dos números naturais não nulos é fácil visualizar a bijeção γ. De fato, se $\gamma(v) = x$ e $v = v_1 v - 2 \cdots v_n$, com os v_i sendo letras em A, e a representação binária de x é $1x_1x_2 \cdots x_m$, com os x_i dígitos binários em $\{0, 1\}$, então temos $n = m$ e $v_i = \mathtt{d}$ sse $x_i = 0$ e $v_i = \mathtt{s}$ sse $x_i = 1$. De fato, veja a representação binária de 1 a 9 e compare com as primeiras palavras de A^* na ordem militar:

$$1, 10, 11, 100, 101, 110, 111, 1000, 1001, \ldots.$$

Esta propriedade pode ser vista como uma extensão das correspondências abaixo, que serão usadas nos algoritmos a seguir:

$$\text{D´} \quad [\mathtt{d}] = 0 \quad \text{e} \quad \text{D´} \quad [\mathtt{s}] = 1,$$

$$\text{L} \quad [0] = \mathtt{d} \quad \text{e} \quad \text{L} \quad [1] = \mathtt{s}.$$

Utilizamos a construção acima através do conceito do "crachá" de um vetor contínuo normalizado que definimos a seguir. Seja $N[1\,..\,n]$ um vetor contínuo e seja v a palavra de comprimento $n - 1$ que representa as diferenças sucessivas de elementos de N. O *crachá* de N é o natural $x = \gamma(v)$. Por exemplo, dado o vetor normalizado em 4.2, temos $n = 6$ e a palavra associada é v, de comprimento 5, dado em 4.3. Substituindo $\mathtt{d}$ por 0 e $\mathtt{s}$ por 1 e acrescentando um dígito mais significativo 1 obtemos 110001, que representa em binário o número 32+16+1=49. O crachá de N é 49.

Inversamente, seja x um natural não nulo e seja $v = \gamma^{-1}(x)$. Seja $n = 1 + |v|$, onde $|v|$ denota o comprimento da palavra v. Associamos a x o vetor contínuo normalizado $N[1\,..\,n]$ de n elementos, que começa com 0 e cujas diferenças sucessivas são exatamente as correspondentes às letras de v. Segue da construção que n e x satisfazem as relações

$$n = 1 + \text{Log}[x] \quad \text{e} \quad 2^{n-1} \leq x < 2^n.$$

Exemplificamos estas definições com a Tabela 4.5 que apresenta os crachás dos vetores normalizados dos 7 blocos do nosso exemplo. O crachá do vetor normalizado do g-ésimo bloco será guardado em C ´$[g]$.

O Algoritmo 4.9, F C ´, calcula o vetor C ´ a partir do vetor V. Os cálculos simplesmente implementam a construção acima delineada. Claramente a complexidade de F C ´ é $O(m)$.

g	1	2	3	4	5	6	7
sd's	ssd	sss	sdd	dss	sds	dsd	dd
C ´$[g]$	14	15	12	11	13	10	4

Tabela 4.5: Os crachás dos 7 blocos do exemplo

Algoritmo 4.9 Cálculo dos crachás dos blocos de V

F C ´ (V, b)

$x \leftarrow 0$
para i **de** 2 **até** m **faça**
    $(q, r) \leftarrow (i - 1)$ **div** b
    se $r = 0$ **então** ▹ bloco completo, começar outro
        C ´$[q] \leftarrow x + \text{Exp}[b - 1]$
        $x \leftarrow 0$
    senão-se $V[i] = V[i - 1] + 1$ **então** ▹ letra s, dígito 1
        $x \leftarrow 2 * x + 1$
    senão $x \leftarrow 2 * x$ ▹ letra d, dígito 0
$(g_m, p_m) \leftarrow$ A (m)
C ´$[g_m] \leftarrow x + \text{Exp}[p_m - 1]$
devolva C ´

Precisaremos, no decorrer dos algoritmos, decompor vetores contínuos normalizados representados por seus crachás. Isto é feito no Algoritmo 4.10 que implementa o procedimento Q (x, c_1). Seja x um crachá que representa o vetor normalizado $N[1\,..\,n]$, com $n \geq 2$, e seja c_1 um inteiro $0 \leq c_1 < n-1$. O vetor N pode ser decomposto em dois vetores não vazios:

$$N[1\,..\,n] = N[1\,..\,c_1+1] \cdot N[c_1+2\,..\,n],$$

com c_1+1 e $n-c_1-1$ elementos, respectivamente. Note que o segundo vetor é contínuo, mas pode não ser normalizado. Seja, portanto, $N'[c_1+2\,..\,n]$ o vetor normalizado de $N[c_1+2\,..\,n]$. Sejam x_1 e x_2 os crachás de $N_1 = N[1\,..\,c_1+1]$ e $N_2 = N'[c_1+2\,..\,n]$, respectivamente. Então, os comprimentos das palavras correspondentes a N_1 e N_2 são, respectivamente, c_1 e $n-c_1-2$. Note que na fronteira $N[c_1+1\,..\,c_1+2]$ temos uma letra em $A = \{\mathsf{s}, \mathsf{d}\}$, correspondente à diferença $N[c_1+2] - N[c_1+1]$ que não influi nem em x_1 nem em x_2. O procedimento Q (x, c_1) devolve uma tripla (x_1, letra, x_2), onde x_1 e x_2 são os crachás de N_1 e N_2, respectivamente, e letra representa a diferença $N[c_1+2] - N[c_1+1]$ que ocorre na fronteira. é a letra na fronteira $N[c_1+1\,..\,c_1+2]$. As operações em que o procedimento se apóia são o quociente e o resto da divisão inteira por potências de 2. Nitidamente, a complexidade de Q é $O(1)$.

Exemplificando esta decomposição, consideremos o vetor normalizado N de 4.2 de crachá 49. Para $c_1 = 3$, a decomposição de N é dada por $N = (0, 1, 0, -1) \cdot (-2, -1)$. O vetor normalizado da segunda parte é $N_2 = (0, 1)$. A letra na fronteira $(-1, -2)$ é d. Os crachás dos pedaços correspondem às palavras sdd e s valem, respectivamente, 1100 e 11 em binário, ou seja, 12 e 3. Assim, Q $(49, 3) = (12, \mathsf{d}, 3)$. É uma aritmética peculiar, mas simples e, principalmente, eficiente. Obtemos todas as informações sobre a decomposição de x em tempo constante.

Algoritmo 4.10 Decomposição de crachás

Q (x, c_1)
$c_2 \leftarrow \mathrm{Log}[x] - c_1 - 1$
$(y_1, y_2) \leftarrow x$ **div** $\mathrm{Exp}[c_2]$
$x_2 \leftarrow y_2 + \mathrm{Exp}[c_2]$
$(x_1, z) \leftarrow y_1$ **div** 2
devolva $(x_1, \mathrm{L}\ \ [z], x_2)$

A técnica dos quatro russos está implementada no procedimento R -, dado no Algoritmo 4.11. Este procedimento calcula um vetor R $[1\,..\,2^b-1]$, com 2^b-1 posições, indexadas pelos crachás de vetores contínuos normalizados com até b elementos. O valor de R $[x]$ é a posição do mínimo no vetor contínuo normalizado de crachá x. O vetor R é calculado usando-se um vetor auxiliar, $r[1\,..\,2^b-1]$. Para um crachá x, seja $N[1\,..\,n]$ o vetor contínuo normalizado de crachá x. Lembre que $n = 1 + \mathrm{Log}[x]$. O elemento $r[x]$ é uma tripla

$$r[x] = (pm, vm, d),$$

onde pm é a posição do mínimo no vetor N, vm é o valor deste mínimo e d é a variação total do vetor N, isto é $d = N[n] - N[1] = N[n]$. Por exemplo, de 4.2 obtemos que $r[49] = (5, -2, -1)$.

Os cálculos são feitos na ordem de crachás crescentes, assim, o cálculo de $r[x]$ pode usar (e usa!) os resultados de vetores mais curtos. Os crachás dos vetores mais curtos são obtidos pelo uso de Q . Nitidamente, a complexidade de R (b) é $O(2^b)$: temos um número constante de operações para cada crachá. Note, para uso futuro, que os coeficientes "escondidos" pela notação $O()$ são pequenos no caso.

Algoritmo 4.11 Cálculo dos Russos de comprimento pequeno

R (b)

$r[1] \leftarrow (1, 0, 0)$
R $[1] \leftarrow 1$
para x **de** 2 **até** $\mathrm{Exp}[b] - 1$ **faça**
  $c \leftarrow \mathrm{Log}[x]$ ▹ c é o comprimento da palavra em $\{\,\mathsf{s}, \mathsf{d}\,\}^*$
  $(x_1, \mathrm{letra}, x_2) \leftarrow$ Q $(x, c-1)$ ▹ volta: $x_2 = 1$
  $(pm, vm, d) \leftarrow r[x_1]$
  $d' \leftarrow d +$ V [letra]
  se $vm \le d'$ **então**
    $r[x] \leftarrow (pm, vm, d')$
  senão $r[x] \leftarrow (c+1, d', d')$
  R $[x] \leftarrow r[x][1]$ ▹ primeira componente de $r[x]$
devolva R

Para exemplificarmos o Algoritmo 4.11, a Tabela 4.6 apresenta os resultados finais de R (4) enquanto a Tabela 4.7 apresenta os resultados intermediários, guardados no vetor $r[x]$.

Note que o procedimento R (*b*) independe do vetor de dados V. O vetor R é uma "constante universal", ele poderia ser armazenada na Internet, por exemplo, e consultado em tempo constante por quem quer que seja. Esta economia de pré-processamento, no entanto, não iria diminuir a complexidade dos nossos algoritmos finais.

x	1	2	3	4	5	6	7
sd's	λ	d	s	dd	ds	sd	ss
R [x]	1	2	1	3	2	1	1

x	8	9	10	11	12	13	14	15
sd's	ddd	dds	dsd	dss	sdd	sds	ssd	sss
R [x]	4	3	2	2	4	1	1	1

Tabela 4.6: Os russos dos 15 crachás de comprimento até 3

x	1	2	3
$r[x]$	$(1,0,0)$	$(2,-1,-1)$	$(1,0,1)$

x	4	5	6	7
$r[x]$	$(3,-2,-2)$	$(2,-1,0)$	$(1,0,0)$	$(1,0,2)$

x	8	9	10	11
$r[x]$	$(4,-3,-3)$	$(3,-2,-1)$	$(2,-1,-1)$	$(2,-1,1)$

x	12	13	14	15
$r[x]$	$(4,-1,-1)$	$(1,0,1)$	$(1,0,1)$	$(1,0,3)$

Tabela 4.7: Vetor de valores auxiliares no cálculo dos russos

O Algoritmo 4.12, M I , tem complexidade $O(1)$. Ele calcula M (i, j) quando i e j caem no mesmo bloco, caso contrário ele devolve 0. Note que 0 é um índice inválido no vetor V e assim este valor é usado para sinalizar que i e j caem em blocos distintos. Este é um expediente que simplifica (um pouco) os programas seguintes.

Algoritmo 4.12 Cálculo do M interno a um bloco

M I (i, j)

▹ i e j são índices de elementos no vetor de dados V
▹ devolvemos 0 se i e j estiverem em blocos distintos
▹ devolvemos M (i, j), caso contrário
$(g_i, p_i) \leftarrow$ A (i)
$(g_j, p_j) \leftarrow$ A (j)
se $g_i \neq g_j$ **ou** $i > j$ **então**
    devolva 0
$(x_1, \text{letra}, x_2) \leftarrow$ Q (C ´$[g_i], p_i - 1)$
$(x_3, \text{letra}, x_4) \leftarrow$ Q $(x_2, j - i)$
devolva $i - 1 +$ R $[x_3]$

O Algoritmo 4.13, T (V, b), calcula a tabela PosMin, versão compacta da (ex) tabelona PosMin. Ele usa M I para calcular as posições dos mínimos de blocos individuais, quanto ao resto, ele é similar ao pré-processamento da seção anterior 4.5.2, Algoritmo 4.6. A complexidade de T (V, b) é $O(m' \log m')$, onde $m' = \lfloor m/b \rfloor$. A nova matriz PosMin, referente ao nosso exemplo está dada na Tabela 4.8.

Algoritmo 4.13 Cálculo das respostas para uniões de blocos

T (V, b)

$m' \leftarrow \lfloor m/b \rfloor$ ▹ Número de blocos inteiros
para i **de** 1 **até** m' **faça**
    PosMin$[i, 0] \leftarrow$ M I $((i-1) * b + 1, i * b)$
para k **de** 1 **até** Log$[m']$ **faça**
    para i **de** 1 **até** $m' -$ Exp$[k]$ **faça**
        PosMin$[i, k] \leftarrow$ P L´ $(i, i + \text{Exp}[k] - 1)$
devolva PosMin

O Algoritmo 4.14, P ´M (V), implementa o pré-processamento da solução linear. Ele calcula logaritmos e potências em tempo $O(m)$. Fixa o valor de b em Log$[m] = \lfloor \log_2 m \rfloor$. Calcula o vetor R em tempo $O(2^b) = O(m)$. Calcula, a seguir, C ´ em tempo $O(m)$. Finalmente,

i	1	2	3	4	5	6
$k = 0$	(1) 1	(2) 5	(3) 12	(4) 14	(5) 17	(6) 22
$k = 1$	(2) 1	(3) 5	(4) 12	(5) 14	(6) 17	
$k = 2$	(4) 1	(5) 5	(6) 12			

Tabela 4.8: A matriz PosMin para o pré-processamento linear

calcula a matriz PosMin em tempo $O(m' \log m') = O(m)$, já que

$$m' = \lfloor m/b \rfloor = \lfloor m/\lfloor \log m \rfloor \rfloor.$$

Queremos realçar que as constantes "escondidas" pela notação $O()$ são todas "pequenas" no caso deste algoritmo. Trata-se de um procedimento rebuscado, mas natural em todos os seus passos. Neste ponto a versão de M que apresentamos difere de todas as outras que conhecemos, já que elas recorrem a artifícios que comprometem a naturalidade do procedimento sem comprometer a sua complexidade. Isto, porém, aumenta, em geral, as constantes "escondidas" pela notação $O()$.

Algoritmo 4.14 Pré-processamento linear para M

```
P ´M (V)
  (Log, Exp) ← L                 EP  ˆ      (m)
  b ← Log[m]
  R       ← R          (b)
  C         ´ ← F  C       ´ (V, b)
  PosMin ← T        (V, b)
```

Finalmente, o procedimento de consulta M (i, j), em tempo constante, é dado no Algoritmo 4.15. Ele é comprido, mas é simples. Inicialmente, define-se V[0] como ∞, para facilitar as contas de mínimos em casos particulares; note que ∞ é o elemento neutro da operação min. Deixamos o comando 1 no Algoritmo 4.15 como um lembrete, nitidamente este comando poderia ser transferido para o pré-processamento. A seguir, "tentamos a sorte" chamando M I (i, j). Se i e j estão em blocos distintos então decompomos a consulta em 3 sub-consultas, respondidas por r_1, r_2 e r_3, uma ou duas das quais podem se referir a mínimos de conjuntos vazios, caso em que o seu resultado será a posição 0. Temos:

- r_1 é a posição do mínimo num intervalo próprio de um eventual bloco inicial, este intervalo é sempre da forma $[i\,..\,b]$;

- r_2 é a posição do mínimo referente à união de blocos inteiros, ele é calculado de forma inteiramente análoga à consulta feita no Algoritmo 4.7, M I ´ (i, j); o cálculo de r_3 chama P L´ , dado no Algoritmo 4.5;

- r_3 é a posição do mínimo num intervalo próprio de um eventual bloco final, este intervalo é sempre da forma $[1, j]$.

As últimas linhas do Algoritmo 4.15 calculam a posição do mínimo de V$[r_1]$, V$[r_2]$ e V$[r_3]$. Esta será a resposta à consulta feita. Nitidamente o Algoritmo 4.15 tem complexidade $O(1)$, observação que encerra esta seção.

4.6 Mac em tempo $\langle O(n), O(1)\rangle$

Dada uma árvore **T**, o seu pré-processamento consiste na execução da busca em profundidade, para trabalharmos, em seguida, com o vetor P das profundidades dos vértices visitados. Como $m = 2n - 1$, a complexidade do pré-processamento é $O(n)$, supondo que estamos usando o pré-processamento linear para M .

Suponhamos, agora, que vértices u e v de **T** sejam tais que D $[u]$ < D $[v]$. Então,

$$\text{M}\quad(u, v) = \text{B}\qquad[\text{M}\quad(\text{D}\quad[u], \text{D}\quad[v])]$$

A consulta M (u, v) é feita em tempo constante.

Desta forma, os algoritmos principais de pré-processamento e de consulta do capítulo são aqueles dados nos Algoritmos 4.16 e 4.17. A complexidade deste par de algoritmos é $\langle O(n), O(1)\rangle$. A sua corretude segue do Teorema 4.2.

Aproveitamos para observar que os Algoritmos 4.16 e 4.17 podem ser usados com qualquer versão da solução para M . Eles provam, na verdade, que dado um par de algoritmos de complexidade $\langle f(m), g(m)\rangle$ para a solução de M , o Menor Ancestral Comum em árvores pode ser encontrado com complexidade $\langle O(n) + f(m), g(m)\rangle$.

Algoritmo 4.15 Consulta M em tempo constante para pré-processamento linear

M (i, j)
V[0] $\leftarrow \infty$ ▹ Anteparo para o cálculo dos mínimos
$r \leftarrow$ M I (i, j) ▹ i e j no mesmo grupo?
se $r \neq 0$ **então**
    devolva r
▹ Calculamos i' e j' satisfazendo $i \leq i'$ e $j' \leq j$
$i' \leftarrow \lceil (i-1)/b \rceil * b + 1$ ▹ i' inicia o primeiro bloco depois de i
$r_1 \leftarrow$ M I $(i, i'-1)$
$j' \leftarrow \lfloor j/b \rfloor * b$ ▹ j' termina o último bloco antes de j
$r_3 \leftarrow$ M I $(j'+1, j)$
▹ $g_{i'}$ e $g_{j'}$ são os blocos contendo i' e j', respectivamente
$g_{i'} \leftarrow 1 + (i'-1)/b$
$g_{j'} \leftarrow j'/b$
se $g_{i'} \leq g_{j'}$ **então**
    $r_2 \leftarrow$ P L´ $(g_{i'}, g_{j'})$
senão $r_2 \leftarrow 0$
▹ Calculamos, a seguir, a posição do mínimo de V$[r_1]$, V$[r_2]$ e V$[r_3]$
se V$[r_1] \leq$ V$[r_2]$ **então**
    $r \leftarrow r_1$
senão $r \leftarrow r_2$
se V$[r_3] <$ V$[r]$ **então**
    $r \leftarrow r_3$
devolva r ▹ r é a posição do menor elemento em V$[i \,..\, j]$

Algoritmo 4.16 Pré-processamento para o cálculo de M

P ´MAC(**T**)
$(m,$ B , P , D , A $) \leftarrow$ B E P (**T**)
V $\leftarrow$ P
P ´M (V)

Algoritmo 4.17 Consulta para M

M (u, v)
se D $[u] \leq$ D $[v]$ **então**
  devolva B [M (D $[u]$, D $[v]$)]
senão **devolva** B [M (D $[v]$, D $[u]$)]

4.7 Notas bibliográficas

O problema do Menor Ancestral Comum em Árvores tem uma longa história da melhoras. A primeira solução não trivial aparece em 1973 e é publicada em [AHU76b]. A primeira solução de complexidade $\langle O(n), O(1)\rangle$ é de 1984 e aparece em Harel e Tarjan [HT84]. Estes algoritmos são muito complexos e foram simplificados em 1988 por Schieber e Vishkin em [SV88] e em 2000 por Bender e Farach-Colton em [BFC00]. O problema do Mínimo de um Vetor aparece em 1984, num trabalho de Gabow, Bentley e Tarjan em [GBT84]. Existe uma descrição detalhada da variante de Shieber e Vishkin no livro de Gusfield [Gus97], que é um dos raros casos em que o algoritmo é descrito com alguns detalhes num livro. Existe uma ampla literatura sobre o problema, seus usos e suas generalizações. Recomendamos a consulta aos trabalhos [BPSS01, AGKR02, BFC02].

Capítulo 5

Aplicações de MAC

Neste capítulo veremos uma aplicação das soluções eficientes dos dois resultados centrais vistos nos Capítulos 2 e 4. Será resolvido de forma eficiente um dos mais importantes e mais recorrentes problemas dentro da Biologia Molecular Computacional, o da busca de padrão com erros.

5.1 O problema da extensão comum

Para uma resolução mais eficiente do problema da busca de padrão com erros, resolveremos primeiramente o sub-problema da extensão comum mais longa definido a seguir: são dadas as palavras s e t, de comprimentos m e n respectivamente, e uma seqüência de pares de posições $(i, j) \in \{1, \ldots, m\} \times \{1, \ldots, m\}$. Para cada um dos pares (i, j) queremos calcular o comprimento do fator comum mais longo de s e de t que ocorre na posição i em s e j em t.

Para resolver o problema de forma eficiente, faremos uso de uma solução eficiente do problema do menor ancestral comum.

A função P E ˜ C do Algoritmo 5.1 faz um pré-processamento das palavras s e t em tempo $O(m + n)$ devido ao Teorema 2.10. Cada chamada de E ˜ C , por sua vez, é calculada em tempo constante. De fato, os cálculos feitos às linhas 3 e 4 podem ser feitas com simples consultas V´ F (s, i) e V´ F (t, j) se esta tabela foi construída como descrito junto ao Algoritmo 2.7. Mesmo que tal tabela não exista, pode-se facilmente construí-la em P E ˜ C através de uma chamada S $(s, 1, \mathbf{T})$ seguida de posterior percurso na lista ligada definida pelas ligações de sufixos. Naturalmente, repete-se semelhante processo para a palavra t. Este eventual custo extra em P E - ˜ C também é linear. Por fim, o cálculo feito à linha 5 também é feito em tempo constante, conforme visto no Capítulo 4.

Assim, o Algoritmo 5.1 é uma solução $\langle O(m+n), O(1)\rangle$ para o problema da extensão comum mais longa.

Algoritmo 5.1 Solução $\langle O(m+n), O(1)\rangle$ da extensão comum mais longa

```
P E ˜ C (s, t)
 1  devolva Á D S ({s, t})

E ˜ C (i, j)
 1  ▹ computa o mais longo prefixo comum de s[i . . |s|] e de t[j . . |t|]
 2  ▹ supõe a árvore dos sufixos de {s, t}
 3  u ← vértice final associado ao sufixo s[i . . |s|]
 4  v ← vértice final associado ao sufixo t[j . . |t|]
 5  r ← M (u, v)                 ▹ menor ancestral comum de u e de v
 6  devolva C (w)                ▹ profundidade de palavra do vértice r
```

5.2 Busca de padrão com erros

Na Seção 3.1 vimos algumas soluções para o importante problema da busca exata de uma palavra (padrão) como fator de outra (texto). Este problema é importante, como vimos em Biologia Molecular Computacional, mas seu uso é limitado desta forma. Um dos principais motivos desta limitação é que várias perturbações podem ocorrer durante o processo biológico de obtenção de uma seqüência (de DNA ou proteína, por exemplo). Outro importante fator é que ao longo do processo evolutivo de uma espécie, vários eventos biológicos ocorrem de forma a introduzir pequenas alterações na seqüência de DNA de uma espécie. Foge ao escopo deste trabalho descrever estes possíveis eventos, mas gostaríamos de comentar que o mais freqüente destes fenômenos é o da mutação, onde um nucleotídio (*a*, *c*, *t* ou *g*) é substituído por outro. Por exemplo, um trecho *actagaatc* que é transformado em *agttgaatg* sofreu três mutações: duas de *c* para *g* nas posições 2 e 9 e de *a* para *t* na posição 4. De forma visual, podemos verificar isto através de um alinhamento das duas seqüências como o abaixo, onde as identidades são representadas pelas barras verticais e as diferenças por espaços em branco na linha intermediária.

```
actagaatc
| |  ||||
agttgaatg
```

Estas considerações motivam a generalização do problema da busca de padrão visto na Seção 3.1.

O problema que pretendemos resolver nesta seção é o problema da busca de padrão com erros agora descrito. São dados uma palavra s de comprimento m, também chamada padrão, uma outra palavra t de comprimento n, também chamada texto e um inteiro $k \geq 0$ fixo. Queremos saber se existe fator de t com o mesmo comprimento de s e cujas letras diferem das de s em no máximo k posições. Tipicamente, k é muito menor que m, que por sua vez é menor que n, i.e.,

$$k \ll m < n.$$

5.2.1 Solução $O(mn)$ para busca de padrão com erros

Apesar do problema ser mais complicado que o da busca de padrão num texto, uma solução ingênua para o problema não é mais complexa que a solução ingênua B I ˆ D P ˜ vista no Algoritmo 3.1. Basta para tanto que que laço à linha 9 não seja interrompido toda vez que uma diferença for detectada à linha 10, mas somente quando o número destas desigualdades superar k. Reescrevendo o algoritmo, temos então o Algoritmo 5.2, cuja complexidade computacional é a mesma $O(mn)$, como imediatamente demonstra-se com a mesma argumentação vista na análise do Algoritmo 3.1.

Apesar desta complexidade não ser pior que a do algoritmo ingênuo de busca exata de padrão, dado que tipicamente temos que $k \ll m$, uma solução $O(kn)$ para o problema seria muito mais eficiente. É o que encontraremos a seguir.

5.2.2 Solução $O(kn)$ para a busca de padrão com erros

O Algoritmo 5.3 é uma aplicação do problema da extensão comum a uma solução $O(kn)$ do problema da busca de padrão com erros. Para cada

---

Algoritmo 5.2 Solução $O(mn)$ para busca de padrão com erros

B I ˆ D P ˜ C E (s, t)

▹ a palavra s é procurada num
▹ texto t com até k letras erradas
▹ devolve $s \in \mathrm{Fat}(t)$ (se $s \in \mathrm{Fat}(t)$ e c.c.)
$r \leftarrow$
$i \leftarrow 0$
enquanto $i \leq |t| - |s|$ **e** $r =$ **faça**
    $r \leftarrow$
    $e \leftarrow 0$
    $j \leftarrow 1$ ▹ testa se $s = t[i + 1 \,..\, |s|]$ a menos de até k erros
    enquanto $j \leq |s|$ **e** $r =$ **faça**
        se $t[i + j] \neq s[j]$ **então**
            $e \leftarrow e + 1$
            se $e > k$ **então**
                $r \leftarrow$
        $j \leftarrow j + 1$
    $i \leftarrow i + 1$
devolva r

---

iteração à linha 6, o algoritmo verifica se $s = t[j+1\,..\,j+|s|]$, a menos de não mais que k letras erradas. Definimos i como sendo o comprimento do prefixo mais comprido de s que se encontra na posição $j+1$ de t, com até e erros. Se o número de erros é nulo, naturalmente que i vale E ˜ C (1, $j+1$), como é calculado à linha 9. Com as iterações do laço à linha 10, enquanto este prefixo não for toda a palavra s e até um limite de k erros, contabilizamos mais um erro $s[i+1] \neq t[j+i+1]$ à linha 11. Este erro a mais expande i através da parcela 1 à linha 12 e a subseqüente parcela E ˜ C $(i+2, j+i+2)$. Observe que todas as operações do

---

Algoritmo 5.3 Solução $O(kn)$ para busca de padrão com erros

B D P ˜ (s,t,k)
1 ▹ verifica se a palavra s pode ser encontrada
2 ▹ num texto t com não mais que k erros
3 **T** $\leftarrow$ P E ˜ C (s,t)
4 $j \leftarrow 0$ ▹ j: deslocamento do padrão s em relação ao texto t
5 $i \leftarrow 0$ ▹ i: o quanto de s foi encontrado na posição $j+1$
6 **enquanto** $i < |s|$ **e** $j \leq |t| - |s|$ **faça**
7 ▹ verifica se $s = t[j+1\,..\,j+|s|]$, com até k letras erradas
8 $e \leftarrow 0$ ▹ e: número de erros cometidos
9 $i \leftarrow$ E ˜ C $(1, j+1)$
10 **enquanto** $i < |s|$ **e** $e < k$ **faça**
11 $e \leftarrow e+1$ ▹ contabiliza erro $s[i+1] \neq t[j+i+1]$
12 $i \leftarrow i+1+$ E ˜ C $(i+2, j+i+2)$
13 $j \leftarrow j+1$
14 **devolva** $i = |s|$ ▹ devolve se e só $i = |s|$

---

algoritmo são realizáveis em tempo constante, a não ser a da linha 3, que é feita em tempo $O(m+n)$. Como são até $n-m+1$ iterações à linha 6 e até k iterações à linha 10, temos que um total de até $k(n-m+1)$ execuções da linha 12 e até $(n-m+1)$ execuções da linha 9 são realizadas. Em cada uma delas, a função E ˜ C computa em tempo $O(1)$ o comprimento de um prefixo comum mais comprido. (Isto revela a importância de uma implementação eficiente desta consulta.) Temos assim uma complexidade computacional $O(k(n-m)+m+n)$ (que é $O(kn)$) para o algoritmo completo.

Capítulo 6

Comparação de Seqüências

Neste capítulo apresentaremos uma resenha da teoria e prática de comparação de seqüências, feita através da obtenção de uma subseqüência comum mais comprida (LCS). A parte prática é feita com foco no diff, a ferramenta UNIX para obtenção de diferenças de arquivos-texto. As seções subseqüentes têm o seu texto em Inglês e são baseadas no artigo [Sim89].

6.1 Sequence comparison

Sequence comparison is a deep and fascinating subject in Computer Science, both theoretical and practical. However, in our opinion, neither the theoretical nor the practical aspects of the problem are well understood and we feel that their mastery is a true challenge for Computer Science.

The central problem can be stated very easily: find an algorithm, as efficient and practical as possible, to compute a longest common subsequence (lcs for short) of two given sequences[1].

As usual, a subsequence of a sequence is another sequence obtained from it by deleting some (not necessarily contiguous) terms. Thus, both `en␣pri` and `en␣pai` are longest common subsequences of `sequence␣comparison` and `theory␣and␣practice`.

It turns out that this problem has many, many applications in many, many apparently unrelated fields, such as computer science, mathema-

[1]The sequences we consider are usually called words. We avoid this terminology since it might lead to confusion because of the widespread misuse of the term subword (meaning a segment or a factor instead of a subsequence).

tics, molecular biology, speech recognition, gas chromatography, bird song analysis, etc. A comprehensive study of the role of the problem in these fields and how it is coped with in each of them can be found in the beautiful book of Sankoff and Kruskal [SK83].

In particular, in computer science the problem has at least two applications. The main one is a file comparison utility, nowadays universally called diff, after its popularization through the UNIX operating system. This tool is intensively used to discover differences between versions of a text file. In this role it is useful in keeping track of the evolution of a document or of a computer program. It is also used as a file compression utility, since many versions of a (long) file can be represented by storing one (long) version of it and many (short) scripts of transforming the stored version in the remaining ones. Another aplication in computer science is to approximate string matching used, for instance, in the detection of misspelled versions of names. It should be noted, however, that sequence comparison is not the main tool to solve this very important problem. For more details the reader is referred to [HD80].

An interesting aspect of the problem is that it can be solved by a simple and perhaps even intuitive 'folklore' algorithm based on a dynamic programming approach. This appealing algorithm has been discovered many, many times. Indeed, it has been discovered by engineers, by biologists and by computer scientists, in Russia, Japan, United States, France and Canada in the period 1968 to 1975. The first publication of the algorithm seems to be, according [SK83], in a 1968 paper by the russian engineer Vintsyuk [Vin68].

The big challenge to computer science comes from the complexity of the folklore algorithm. Indeed, it requires time proportional to the product of the lengths of the sequences, and no essentially better practical algorithm is known. The question is to search for a possible algorithm which is simultaneously efficient and practical.

As far as we know, the existence of a linear algorithm has not been ruled out. Neither has been found a practical algorithm which worst case time complexity is better than $O(mn)$, where m and n are the lengths of the given sequences. There exists, however, an algorithm of time complexity $O(n^2/\log n)$ for pairs of sequences of length n over a fixed finite alphabet, discovered by Masek and Paterson [MP80]. This algorithm is not suitable for practical purposes but its existence is a hint that better algorithms than the ones in current use must exist.

All told, a very good start would be to find out whether or not there exists a linear time algorithm to compute a longest common subsequence of two given sequences over two letters. We get a more modest but still very interesting start by replacing "linear time" for "time complexity $O(n \log n)$". Here, the time bounds should be measured on a random access machine under the unit-cost model.

A weaker version, already answered by Masek and Paterson, has been proposed by D. E. Knuth in a technical report coauthored with V. Chvátal and D. A. Klarner in 1972 [CKK72]:

```
Problem 35.   Greatest common substrings.
  It is possible to find the longest common subsequence of two
sequences of a's and b's in a time proportional to the product
of their lengths. Can one do better?
Note: aba is a subsequence of aabbbba.
```

Incidentally, this seems to be the first reference to the problem and to the folklore algorithm within Computer Science.

6.2 Some theory

We begin this section with the presentation of the folklore algorithm which is the starting point for most of the known algorithms to find an lcs of two given sequences.

Let $u = u_1 u_2 \cdots u_n$ and $v = v_1 v_2 \cdots v_m$ be sequences of lengths n and m over an alphabet A. We assume that each u_i and v_j is a letter in A. The folklore algorithm consists of computing the length $d(i, j)$ of an lcs of $u_1 \cdots u_i$ and $v_1 \cdots v_j$. This can be done by observing that $d(0, 0) = 0$ and by repeatedly applying the formula:

$$d(i, j) = \begin{cases} 1 + d(i-1, j-1) & \text{if } u_i = v_j, \\ \max\{\, d(i-1, j), d(i, j-1) \,\} & \text{if } u_i \neq v_j. \end{cases}$$

Having computed the matrix d the length of an lcs of u and v is $d(n, m)$ and a common subsequence of length $d(n, m)$ can be computed easily proceeding backwards: from $d(n, m)$ to $d(0, 0)$. Figure 6.1 shows an application of the folklore algorithm.

	s	e	q	u	e	n	c	e	␣	c	o	m	p	a	r	i	s	o	n
t	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
h	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
e	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
o	0	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2
r	0	1	1	1	1	1	1	1	1	1	2	2	2	2	3	3	3	3	3
y	0	1	1	1	1	1	1	1	1	1	2	2	2	2	3	3	3	3	3
␣	0	1	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3
a	0	1	1	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3	3
n	0	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3	4
d	0	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3	4
␣	0	1	1	1	1	2	2	2	3	3	3	3	3	3	3	3	3	3	4
p	0	1	1	1	1	2	2	2	3	3	3	3	4	4	4	4	4	4	4
r	0	1	1	1	1	2	2	2	3	3	3	3	4	4	5	5	5	5	5
a	0	1	1	1	1	2	2	2	3	3	3	3	4	5	5	5	5	5	5
c	0	1	1	1	1	2	3	3	3	4	4	4	4	5	5	5	5	5	5
t	0	1	1	1	1	2	3	3	3	4	4	4	4	5	5	5	5	5	5
i	0	1	1	1	1	2	3	3	3	4	4	4	4	5	5	6	6	6	6
c	0	1	1	1	1	2	3	3	3	4	4	4	4	5	5	6	6	6	6
e	0	1	1	1	2	2	3	4	4	4	4	4	4	5	5	6	6	6	6

Figura 6.1: An example for the folklore algorithm

The time complexity of this algorithm is clearly $O(nm)$; actually this time does not depend on the sequences u and v themselves but only on their lengths. By choosing carefully the order of computing the $d(i, j)$'s one can execute the above algorithm in space $O(n + m)$. Even an lcs can be obtained within this time and space complexities but this requires a clever subdivision of the problem [Hir75].

An important property of this dynamic programming algorithm is that it can easily be generalized to computing the minimum cost of editing u into v given (possibly different) costs of the operations: changing a letter into another and deleting or inserting a letter [WF74].

The literature contains a large number of variants of this algorithm, most of them are recorded in the bibliography. Just to give some idea about the various time bounds Table 6.1 lists a few results, where we assume that

u and v have the same length n. In the table p denotes the length of the result (an lcs of u and v). Also, r denotes the number of matches, i.e. the number of pairs $(i, j) \in [1, n] \times [1, n]$ for which $u_i = v_j$.

Hunt and Szymanski(77) [HS77]	$O((r + n) \log n)$
Hirschberg(77) [Hir77]	$O(pn + n \log n)$
Hirschberg(77) [Hir77]	$O(((n + 1 - p)p \log n)$
Nakatsu et al.(82) [NKY82]	$O(n(n - p))$
Hebrard(84) [Heb84]	$O(pn)$

Tabela 6.1: Time complexities of some lcs algorithms

None of the algorithms in Table 6.1 have worst case time complexity better than $O(n^2)$, some are even worse. This can be seen by observing that the value of p varies between 0 and n, while that of r varies between 0 and n^2 and their average value, for pairs of sequences over a fixed alphabet, is proportional, respectively, to n and n^2. It is important to note, however, that, for particular cases, some of the algorithms might use considerably less time than in the worst case. A lively description, from a unified viewpoint, of two of the main algorithms and some variations can be found in [AG87], a paper by Apostolico and Guerra.

The most interesting theoretical result is that of Masek and Paterson [MP80]. Carefully using subdivision techniques similar to the ones used in the "Four russian's algorithm" they transformed the folklore algorithm into one with time complexity $O(n^2 / \log n)$. This is indeed the only evidence that there exist faster algorithms than the folklore one.

Before talking about lower bounds we would like to clarify the model we think is appropriate for the lcs problem. First, the time complexity should be measured on a random access machine under the unit cost model. This seems to be the correct model because we are interested in practical algorithms and this is the closest we can get to existing computers. Second, the time complexity should be measured in terms of the total size, say t, of the input, instead of simply considering the length, say n, of the input sequences. This is a delicate point. For sequences over a known and fixed alphabet we can consider $t = n$ and this was the case considered until now. The other common assumption is to consider sequences over a potentially unbounded alphabet. In this case we assume that the letters are coded over some known, fixed and finite auxiliary alphabet; hence, to represent n different symbols we need size $t \in \Omega(n \log n)$. Thus, measuring complexity in terms of n or t turn out to be very different!

This model adjusts very well to the current philosophy of text files whenever each line is considered as a letter. In particular, this is the case of the file comparison utilities, our main example for the unbounded alphabet model. In essence we propose that in this case complexity should be measured in terms of the length of the files instead of their number of lines. The main consequence of this proposal is that it increases considerably, but within reasonable bounds, the world of linear algorithms: just for one example, sorting the lines of a text file can be done in linear time [AHU74, Meh84].

The existing lower bounds for the complexity of the longest common subsequence problem [Fre75, AHU76a, WC76, Hir78] are based on restricted models of computations and do not apply to the model we just proposed. This point seems to be responsible for a certain amount of confusion because sometimes the known lower bounds tend to be interpreted outside the model for which they were obtained. Indeed, as far as we are aware of, no known lower bound excludes the existence of a linear algorithm in the sense just outlined.

Another very interesting, apparently difficult and little developed area is the probabilistic analysis of the quantities envolved in the lcs problem. Let $f(n,k)$ be the average length of the lcs of two sequences of length n over an alphabet A of k letters (the uniform distribution on A^n is assumed). The function $f(n,k)$ has been explicitly computed for small values of n and k in [CS75b]. On the asymptotic side it is known that for every k there exists a constant c_k such that

$$\lim_{n\to\infty} \frac{f(n,k)}{n} = \sup_n \frac{f(n,k)}{n} = c_k.$$

Thus, fixing the finite alphabet A, the length of an lcs of two random sequences in A^n is ultimately proportional to n.

The exact determination of c_k seems elusive and only lower and upper bounds are known for small values of k. Some results in this direction appear in Table 6.2. For more details, see [SK83, Dek79, CS75a].

An interesting conjecture was made by Sankoff and Mainville [SK83]:

$$\lim_{k\to\infty} \sqrt{k}c_k = 2.$$

k	lower bound	upper bound
2	0.76	0.86
5	0.51	0.67
10	0.40	0.54
15	0.32	0.46

Tabela 6.2: Some upper and lower bounds for c_k

We close this section mentioning five results related, in one way or another, to the lcs problem.

A very important subproblem is obtained by restricting the input to permutations (sequences in which each letter occurs at most once). This case was solved by Szymansky [Szy75] in time $O(n \log n)$. Such an algorithm is also contained in work of Hunt and Szymanski [HS77] and that of Hunt and McIlroy [HM76]. It is an open problem whether or not the case of permutations can be done in linear time on the model we proposed.

A further restriction leads to yet another very important subproblem. This is obtained if we consider $(1, 2, \ldots, n)$ as one of the permutations, assuming, of course, the alphabet $[1, n]$. Then an lcs is just a longest increasing subsequence of the second permutation and this problem is part of a very rich theory of representations of the symmetric group using Young tableaux extensively studied by A. Young, G de B. Robinson, C. E. Schensted and M. P. Schützenberger. A survey focusing on the computational aspects can be found in Knuth's book [Knu73] from which Fredman [Fre75] extracted an algorithm to solve the longest increasing subsequence problem. His algorithm runs in time $O(n \log n)$ and he also derives $n \log n$ as a lower bound for the problem. But, beware, the lower bound does not apply to our model! Using the theory of Young tablaux one can compute the number of permutations of $[1, n]$ which has a longest increasing subsequence of any given length. Extensive calculations can be found in [BB68]. However, the expected value of the length of a longest increasing subsequence of a permutation of length n is not known but the data compiled in [BB68] indicate that this value is approximately $2\sqrt{n}$.

The third related problem is obtained if we look for a longest common segment of two sequences instead of a longest common subsequence. In [Cro87] a linear algorithm was obtained to solve this problem.

The fourth related problem is obtained by considering mini-max duality: instead of looking for the longest common subsequence what about a shortest uncommon subsequence? In 1984 the author solved this problem with an algorithm of time complexity $O(|A| + |u| + |v|)$. More precisely, this (unpublished) linear algorithm computes a shortest sequence which distinguishes the sequences u and v over the alphabet A, that is to say, a shortest sequence which is a subsequence of exactly one of the unequal sequences u and v. For instance, consider `sequence␣comparison` and `theory␣and␣practice`: `eee` distinguishes them while `cc` does not. A shortest distinguisher is given by `d`.

The last related problem is a negative result (from our point of view). It was shown by Maier [Mai77] that deciding whether or not a finite set of sequences has a common subsequence of length k is an NP-complete problem. Other related NP-complete problems can be found in [GMS80].

6.3 Some practice

In this section we restrict ourselves to some practical aspects of file comparison utilities, focusing especially on the UNIX command diff.

A file comparison utility should determine the differences between two text files. But what is the difference of two text files? Indeed, our intuitive notion of such a difference is an elusive concept and it is difficult to define it. To cope with this it became common practice to consider entire lines as indivisible objects. Then it seems that the best results are obtained if one finds a longest common subsequence of lines and then anything not in this lcs is declared a difference.

The first (and still the best) file comparison utility was included in UNIX around 1976. Since then file comparison became a standard tool and with the proliferation of microcomputers many programs turned up. These are usually called diff, but most of them do not determine a true lcs; consequently they easily missynchronize. On the other hand, some of them are very fast and work well for many pairs of (real text) files. Many of these programs are based on [Hec78].

One aspect of the file comparison programs for microcomputers is worth mentioning: their output is sometimes more suggestive (for a human) of the differences than the output of the original diff. Indeed, a good way of pinpoiting the differences seems to be a simultaneous listing of both files, indicating whether each line is common to both or exclusive to one of them. Long blocks of lines in the same class might be abbreviated by showing only their first and last lines. In contrast, the output of diff is thoroughly influenced by the intricacies of machine transformation of one file in another and this restricts, in our opinion, its potential as a tool for remembering or discovering the changes during the evolution of a file.

The algorithm actually used by diff is described by Hunt and McIlroy in [HM76]; its basic idea is attributed to unpublished work of H. S. Stone who generalized an $O(n \log n)$ solution of the most important particular case (the restriction of the problem to permutations) by T. G. Szymanski [Szy75]. The resulting algorithm is very similar to the one in [HS77]; it is also described in [AHU83].

The first practical concession of diff is that it hashes the lines of the files. This is handy because it reduces significantly the volume of information to deal with. On the other hand, the hashing might introduce false matches caused by collisions; these are detected during the last phase when the computed lcs is checked in the files themselves. If false matches occur the corresponding lines are considered as differences. Consequently, it might happen that the reported common subsequence of lines is not a longest one. These events seems to be very rare in practice and the advantages of hashing greatly outweight its shortcomings.

The key concept in the algorithm is that of a k-candidate. Returning to our notations in the previous section, a *k-candidate* is a pair of positions (i, j) such that $u_i = v_j$ and

$$k = d(i, j) = d(i, j-1) + 1 = d(i-1, j) + 1 = d(i-1, j-1) + 1.$$

It follows that every lcs of $u_1 \ldots u_i$ and $v_1 \ldots v_j$ is the concatenation of an lcs of $u_1 \ldots u_{i-1}$ and $v_1 \ldots v_{j-1}$ with the letter $u_i = v_j$. The set of k-candidates in Figure 6.1 is

$$(3,2), (9,6), (7,9), (4,11), (15,7), (11,9), (8,14), (5,15),$$
$$(19,8), (15,10), (12,13), (9,19), (14,14), (13,15), (17,16).$$

The basic strategy of the algorithm is to compute the set of all k-candidates and then collect an lcs from these (such an lcs clearly exists). The computation is done by performing a binary search, in a vector of at most n components, for certain matches, that is to say, pairs (i, j) for which $u_i = v_j$. Thus, r being the total number of matches this part of the algorithm takes time $O(r \log n)$ (we assume throughout that both input sequences have length n). The overall worst case time complexity of the algorithm is $O((r + n) \log n)$ and its space requirements are $O(q + n)$, where q is the total number of k-candidates encountered.

A key question is to investigate the total number q of candidates for particular pairs of sequences. This is interesting because $q \log n$ and q are lower bounds for the computing time and for the space requirements, once the present strategy is adopted. Unfortunately, there are pairs, such as $(abc)^n$ and $(acb)^n$ or $(abab)^n$ and $(abba)^n$ for which $q \in \Theta(n^2)$. Consequently, the derived upper bound can be obtained and the complexity of the algorithm is indeed $\Theta(n^2 \log n)$, that is the worst case behavior is even worse than that of the folklore algorithm.

The great advantage of this algorithm is that in the case of permutations the number r of matches is at most n, hence the algorthm works in time $O(n \log n)$. In actual practice the behavior of the algorithm is somewhere between these two bounds. Fortunately, most lines of true text files are either unique, or occur few times; hence, in practice this algorithm is definitely sub-quadratic! And this is why the algorithm works well even for long files (tens of tousands of lines).

One shortcoming of the algorithm of diff is that for families of pairs of sequences with $r \in \Theta(n^2)$ the running time is $\Theta(n^2 \log n)$ even if $q \in \Theta(n)$. There is at least one such family of files which occur in practice and for which diff behaves badly. These are files with many occurrences of one same line, say one fourth of the lines are blank. The easiest misbehavior of diff can be obtained by running it on sequences of the form ab^na and b^n [MM85].

Another shortcoming is that the computing time might depend on the order of specification of the sequences. Thus, computing the diff of $ab^{2n}a$ and b^n takes much longer than the diff of b^n and $ab^{2n}a$. The fact that the difference is not a symmetrical function does not justify this behavior because what dominates the running time is the computation of an lcs and an lcs does not depend on the order of the sequences.

Both these shortcomings disappear in an interesting variant discovered by Apostolico in [Apo86]; see also [AG87]. This variant has time complexity $O((q+n) \log n)$ instead of $O((r+n) \log n)$. This is sufficient to guarantee an $O(n \log n)$ behavior, instead of $O(n^2 \log n)$, for files with only one frequent line, such as the example given above. However, the gain is obtained at the expense of complicated data structures, such as balanced binary search trees, and it is not clear whether the overhead of (always) using these structures is worth the time economy which is more accentuate only for special cases. Some experimentation might throw interesting light on this question.

A family which seems to defeat every known algorithm is given by pairs of random sequences over two letters. These seem to be the real "black sheeps" for sequence comparison; our luck is that they do not occur in practice very frequently. Or, do they? For instance, files that have many occurrences of two different lines in interlaced positions tend to behave as random sequences over two letters. These cases might arise in practice if we have blank lines with different indentations or two lines which occur frequently, such as the pairs **begin** and **end**.

Altogether, in spite of the excellence of diff, there seems to be ample space for a substantially better algorithm, if only it could be found! But we are hopeful that the proliferation of potentially equivalent quadratic algorithms is a sign that the ultimate word was not yet said.

Referências Bibliográficas

[AC75] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Comm. ACM*, 18:333–340, 1975.

[AG87] A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2:315–336, 1987.

[AGKR02] S. Alstrup, C. Gavoille, H. Kaplan, and T. Raulie. Nearest common ancestors: A survey and a new distributed algorithm. In *Proceedings of the 14th Annual ACM Symposium on Parallel ALgorithms and Architectures (SPAA-02)*, pages 258–264, New York, August 10–13 2002. ACM Press.

[AGM$^+$90] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, Oct 1990.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Pu. Co., Reading, MA, 1974.

[AHU76a] A. V. Aho, D. S. Hirschberg, and J. D. Ullman. Bounds on the complexity of the longest common subsequence problem. *Journal of the ACM*, 23:1–12, 1976.

[AHU76b] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. On finding lowest common ancestors in trees. *SIAM J. Comput.*, 5:115–132, 1976.

[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley Pu. Co., Reading, MA, 1983.

[AMS+97] Altschul, Madden, Schäffer, Zhang, Zhang, Miller, and Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, Sep 1997. Review.

[Apo85] Alberto Apostolico. The myriad virtues of subword trees. In Alberto Apostolico and Zvi Galil, editors, *Combinatorial Algorithms on Words*, pages 85–96, Berlin, 1985. Springer-Verlag. NATO Advanced Science Institutes, Series F, Vol. 12.

[Apo86] Alberto Apostolico. Improving the worst-case performance of the Hunt-Szymanski strategy for the longest common subsequence of two strings. *Information Processing Letters*, 23:63–69, 1986.

[BB68] R. M. Baer and P. Brock. Natural sorting over permutation spaces. *Math. Comp.*, 22:385–410, 1968.

[BBH+85] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoret. Comput. Sci.*, 40(1):31–55, 1985. Special issue: Eleventh international colloquium on automata, languages and programming (Antwerp, 1984).

[BFC00] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Latin American Theoretical INformatics*, pages 88–94, 2000.

[BFC02] Michael A. Bender and Martín Farach-Colton. The level ancestor problem simplified. In *LATIN 2002: Theoretical informatics (Cancun)*, volume 2286 of *Lecture Notes in Comput. Sci.*, pages 508–515. Springer, Berlin, 2002.

[BM77] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1977.

[BPM+00] S. Batzoglou, L. Pachter, J. P. Mesirov, B. Berger, and E. S. Lander. Human and mouse gene structure: comparative analysis and application to exon prediction. *Genome Research*, 10(7):950–958, Jul 2000.

[BPSS01] Michael A. Bender, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Finding least common ancestors in directed acyclic graphs. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-01)*, pages 845–854, New York, January 7–9 2001. ACM Press.

[cit] Citeseer. See <http://www.citeseer.com>.

[CKK72] V. Chvátal, D. A. Klarner, and D. E. Knuth. Selected combinatorial research problems. Technical Report STAN-CS-72-292, Computer Science Department, Stanford University, 1972.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1990.

[CR94] Maxime Crochemore and Wojciech Rytter. *Text algorithms*. The Clarendon Press Oxford University Press, New York, 1994. With a preface by Zvi Galil.

[Cro86] Maxime Crochemore. Transducers and repetitions. *Theoret. Comput. Sci.*, 45(1):63–86, 1986.

[Cro87] Maxime Crochemore. Longest common factor of two words. In *Proceedings of CAAP'87, Pisa, Italy*, pages 26–36, 1987.

[CS75a] V. Chvátal and D. Sankoff. Longest common subsequences of random sequences. Technical Report STAN-CS-75-477, Computer Science Department, Stanford University, 1975.

[CS75b] V. Chvátal and D. Sankoff. Longest common subsequences of two random sequences. *J. Appl. Prob.*, 12:306–315, 1975.

[CV97a] Maxime Crochemore and Renaud Vérin. Direct construction of compact directed acyclic word graphs. In *Combinatorial pattern matching (Aarhus, 1997)*, volume 1264 of *Lecture Notes in Comput. Sci.*, pages 116–129. Springer, Berlin, 1997.

[CV97b] Maxime Crochemore and Renaud Vérin. On compact directed acyclic word graphs. In *Structures in logic and computer science*, volume 1261 of *Lecture Notes in Comput. Sci.*, pages 192–211. Springer, Berlin, 1997.

[Dek79] Joseph Deken. Some limit results for longest common subsequences. *Discrete Math.*, 26:17–31, 1979.

[DKF+99] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, Jun 1999.

[dL96] Alair Pereira do Lago. On the Burnside semigroups $x^n = x^{n+m}$. *Internat. J. Algebra Comput.*, 6(2):179–227, 1996.

[dL02] Alair Pereira do Lago. Local groups in free groupoids satisfying certain monoid identities. *Internat. J. Algebra Comput.*, 12(1&2):357–369, 2002.

[dLS01] Alair Pereira do Lago and Imre Simon. Free Burnside semigroups. *Theoretical Informatics and Applications*, 35(6):579–595, 2001.

[DPCS02] Delcher, Phillippy, Carlton, and Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, 30(11):2478–2483, Jun 2002.

[Fre75] M. L. Fredman. On computing the length of longest increasing subsequences. *Discrete Math.*, 11:29–35, 1975.

[GBT84] Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In ACM, editor, *Proceedings of the sixteenth annual ACM Symposium on Theory of Computing, Washington, DC, April 30–May 2, 1984*, pages 135–143, New York, NY, USA, 1984. ACM Press.

[GMS80] John Gallant, David Maier, and James A. Storer. On finding minimal length superstrings. *Journal of Computer and System Sciences*, 20:50–58, 1980.

[goo] Google. See <http://www.google.com>.

[Gus97] Dan Gusfield. *Algorithms on strings, trees, and sequences*. Cambridge University Press, Cambridge, 1997. Computer science and computational biology.

[HD80] P. A. V. Hall and G. R. Dowling. Approximate string matching. *ACM Computing Surveys*, 12:381–402, 1980.

[Heb84] Jean Jacques Hebrard. Distances sur les mots. Application à la recherche de motifs. Thèse de 3e cycle, Université de Haute-Normandie, 1984.

[Hec78] P. Heckel. A technique for isolating differences between files. *Communications of the ACM*, 21:264–268, 1978.

[Hir75] Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18:341–343, 1975.

[Hir77] Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM*, 24:664–675, 1977.

[Hir78] Daniel S. Hirschberg. An information theoretic lower bound for the longest common subsequence problem. *Information Processing Letters*, 7:40–41, 1978.

[HM76] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical Report #41, Computing Science, Bell Laboratories, 1976.

[HS77] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20:350–353, 1977.

[HT84] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[Kar93] Ricardo Ueda Karpischek. O autômato dos sufixos. Master's thesis, Instituto de Matemática e Estatística da Universidade de São Paulo, São Paulo, 1993.

[KCO+01] Kurtz, Choudhuri, Ohlebusch, Schleiermacher, Stoye, and Giegerich. REPuter: the manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Research*, 29(22):4633–4642, Nov 2001.

[KMP77] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:323–350, 1977.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Vol. 3, Sorting and Searching*. Addison-Wesley Pu. Co., Reading, MA, 1973.

[Kur99] Stefan Kurtz. Reducing the space requirement of suffix trees. *Software, Practice and Experience*, 29(13):1149–1171, 1999.

[Les97] Michael Lesk. *Practical Digital Libraries : Books, Bytes, and Bucks*. Morgan Kaufmann, 1997.

[Lot83] M. Lothaire. *Combinatorics on words*. Addison-Wesley Publishing Co., Reading, Mass., 1983. A collective work by Dominique Perrin, Jean Berstel, Christian Choffrut, Robert Cori, Dominique Foata, Jean Eric Pin, Guiseppe Pirillo, Christophe Reutenauer, Marcel-P. Schützenberger, Jacques Sakarovitch and Imre Simon, With a foreword by Roger Lyndon, Edited and with a preface by Perrin.

[Mai77] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25:322–336, 1977.

[McC76] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. Assoc. Comput. Mach.*, 23(2):262–272, 1976.

[Meh84] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag, Berlin, 1984.

[MM85] W. Miller and E. W. Myers. A file comparison program. *Software - Practice and Experience*, 15:1025–1040, 1985.

[MP80] W. J. Masek and M. S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20:18–31, 1980.

[MS97] João Meidanis and João Carlos Setubal. *Introduction to Computational Molecular Biology*. PWS Publishing Company,., Boston, 1997.

[Mye92] Gene Myers. A Four Russians algorithm for regular expression pattern matching. *J. Assoc. Comput. Mach.*, 39(2):430–448, 1992.

[NKY82] N. Nakatsu, Y. Kambayashi, and S. Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica*, 18:171–179, 1982.

[NMWP99] C.G. Nevill-Manning, I.H. Witten, and G.W. Paynter. Lexically-generated subject hierarchies for browsing large collections. *International Journal of Digital Libraries*, pages 111–123, 1999.

[Pea00] Pearson. Flexible sequence similarity searching with the FASTA3 program package. *Methods in Molecular Biology*, 132:185–219, 2000.

[Pev00] Pavel A. Pevzner. *Computational molecular biology*. Computational Molecular Biology. MIT Press, Cambridge, MA, 2000. An algorithmic approach, A Bradford Book.

[Pin86] Jean Eric Pin. *Varieties of Formal Languages*. North Oxford Academic, London, 1986. English translation of a french edition.

[PL88] Pearson and Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of USA*, 85(8):2444–2448, Apr 1988.

[RS97a] G. Rozenberg and A. Salomaa, editors. *Handbook of formal languages. Vol. 1*. Springer-Verlag, Berlin, 1997. Word, language, grammar.

[RS97b] G. Rozenberg and A. Salomaa, editors. *Handbook of formal languages. Vol. 2*. Springer-Verlag, Berlin, 1997. Linear modeling: background and application.

[RS97c] G. Rozenberg and A. Salomaa, editors. *Handbook of formal languages. Vol. 3*. Springer-Verlag, Berlin, 1997. Beyond words.

[Sim89] Imre Simon. Sequence comparison: some theory and some practice. In M. Gross and Dominique Perrin, editors, *Electronic Dictionaries and Automata in Computational Linguistics*, pages 79–92, Berlin, 1989. Springer-Verlag. Lecture Notes in Computer Science, 377.

[SK83] David Sankoff and Joseph B. Kruskal. *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*. Addison-Wesley Pu. Co., Reading, MA, 1983.

[SV88] Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17:1253–1262, 1988.

[Szy75] T. G. Szymanski. A special case of the maximal common subsequence problem. Technical Report TR-170, Computer Science Lab., Princeton University, 1975.

[Tho68] K. Thompson. Regular expression search algorithm. *Comm. ACM*, 11(6):419–422, 1968.

[TM99] T. A. Tatusova and T. L. Madden. BLAST 2 Sequences, a new tool for comparing protein and nucleotide sequences. *FEMS Microbiology Letters*, 174(2):247–250, May 1999.

[Ukk95] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[VAM+01] Venter, Adams, Myers, Li, Mural, Sutton, Smith, Yandell, Evans, Holt, Gocayne, Amanatides, Ballew, Huson, Wortman, Zhang, Kodira, Zheng, Chen, Skupski, Subramanian, Thomas, Zhang, Gabor Miklos, Nelson, Broder, Clark, Nadeau, McKusick, Zinder, Levine, Roberts, Simon, Slayman, Hunkapiller, Bolanos, Delcher, Dew, Fasulo, Flanigan, Florea, Halpern, Hannenhalli, Kravitz, Levy, Mobarry, Reinert, Remington, Abu-Threideh, Beasley, Biddick, Bonazzi, Brandon, Cargill, Chandramouliswaran, Charlab, Chaturvedi, Deng, Di Francesco, Dunn, Eilbeck, Evangelista, Gabrielian, Gan, Ge, Gong, Gu, Guan, Heiman, Higgins, Ji, Ke, Ketchum, Lai, Lei, Li, Li, Liang, Lin, Lu, Merkulov, Milshina, Moore, Naik, Narayan, Neelam, Nusskern, Rusch, Salzberg, Shao, Shue, Sun, Wang, Wang, Wang, Wang, Wei, Wides, Xiao, Yan, Yao, Ye, Zhan, Zhang, Zhang, Zhao, Zheng, Zhong, Zhong, Zhu, Zhao, Gilbert, Baumhueter, Spier, Carter, Cravchik, Woodage, Ali, An, Awe, Baldwin, Baden, Barnstead, Barrow, Beeson, Busam, Carver, Center, Cheng, Curry, Danaher, Davenport, Desilets, Dietz, Dodson, Doup, Ferriera, Garg, Gluecksmann, Hart, Haynes, Haynes, Heiner, Hladun, Hostin, Houck, Howland, Ibegwam, Johnson, Kalush, Kline, Koduru, Love, Mann, May, McCawley, McIntosh, McMullen, Moy, Moy, Murphy, Nelson, Pfannkoch, Pratts, Puri, Qureshi, Reardon, Rodriguez, Rogers, Romblad, Ruhfel, Scott, Sitter, Smallwood, Stewart, Strong, Suh, Thomas, Tint, Tse, Vech, Wang, Wetter, Williams, Williams, Windsor, Winn-Deen, Wolfe, Zaveri, Zaveri, Abril, Guigó, Campbell, Sjolander, Karlak, Kejariwal, Mi, Lazareva, Hatton, Narechania, Diemer, Muruganujan, Guo, Sato, Bafna, Istrail, Lippert, Schwartz, Walenz, Yooseph, Allen, Basu, Baxendale, Blick, Caminha, Carnes-Stine, Caulk, Chiang, Coyne, Dahlke, Mays, Dombroski, Donnelly, Ely, Esparham, Fosler, Gire, Glanowski, Glasser, Glodek, Gorokhov, Graham, Gropman, Harris, Heil, Henderson, Hoover, Jennings, Jordan, Jordan, Kasha, Kagan, Kraft, Levitsky, Lewis, Liu, Lopez, Ma, Majoros, McDaniel, Murphy, Newman, Nguyen, Nguyen, Nodell, Pan, Peck, Peterson, Rowe, Sanders, Scott, Simpson, Smith, Sprague, Stockwell, Turner, Venter, Wang, Wen, Wu, Wu, Xia, Zandieh, and Zhu. The sequence of the human genome. *Science*, 291(5507):1304–1351, Feb 2001.

[Vin68] T. K. Vintsyuk. Speech discrimination by dynamic programming. *Kibernetika*, 4:81–88, 1968.

[WC76] C. K. Wong and A. K. Chandra. Bounds for the string editing problem. *Journal of the ACM*, 23:13–16, 1976.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *14th Annual IEEE Symposium on Switching and Automata Theory (Univ.*

Iowa, Iowa City, Iowa, 1973), pages 1–11. IEEE Comput. Soc., Northridge, Calif., 1973.

[WF74] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21:168–173, 1974.