Aplicações de uma solução eficiente para o problema do Menor Ancestral Comum

Aluno: Alexandre L J H Albano
Orientador: Prof. Dr. Alair Pereira do Lago

Monografia baseada em uma Iniciação Científica
IME USP 2006

Sumário

Capítulo 1

Notações preliminares

Para podermos resumir de maneira breve e explicitar o objetivo deste texto, será necessário primeiramente introduzir a notação utilizada no desenvolvimento deste trabalho. Também iremos descrever o problema do Menor Ancestral Comum antes de exibirmos uma síntese geral.

1.1 Alfabetos e palavras

Definição 1 *Um alfabeto é um conjunto finito, não vazio, cujos elementos serão chamados de símbolos.*

Exemplo 2 $\Sigma = \{a, b, c, d, e, \ldots, v, x, z\}$

Definição 3 *Uma palavra sobre um alfabeto Σ é uma seqüência finita (possivelmente vazia) de elementos de Σ. No caso da seqüência vazia, a palavra será denotada por 1.*

Notação 4 *Se u, w são palavras tais que u é prefixo de w, denotamos por $u^{-1}w$ a palavra obtida pela remoção de u em w.*

Notação 5 *O conjunto das palavras sobre um alfabeto A será denotado A^*. O conjunto $A^* \setminus \{1\}$ será denotado A^+.*

Exemplo 6 *Se $\Sigma = \{a, b, n\}$, então exemplos de palavra sobre Σ são: $banana$, $naba$, aba.*

1.2 Grafos, árvores

Introduzimos agora a notação apropriada para o tratamento de grafos e árvores. Omitiremos as definições básicas sobre grafos, na certeza que o leitor certamente já está bastante familiarizado com tais definições.

Notação 7 *Um grafo orientado* $\overrightarrow{G}$ *será denotado por* $\overrightarrow{G} = (V, E)$*, onde* V *é o conjunto dos vértices de* $\overrightarrow{G}$ *e* E *é o conjunto das arestas de* $\overrightarrow{G}$*.*

Notação 8 *Um caminho em um grafo* $\overrightarrow{G}$ *será denotado por* $< v_1, v_2, \ldots, v_n >$

Notação 9 *Uma árvore* T *será denotada por* $T = (V, E)$*, e é um grafo conexo tal que, existe um vértice* $Raiz(T)$ *que satisfaz a seguinte propriedade: para todo vértice* $v \in V$*, tem-se que o caminho de* $Raiz(T)$ *a* v *existe e é único.*

1.3 Vetores

Notação 10 *Um vetor* V *de tamanho* n *será denotado por* $V[1 \ldots n]$*. Naturalmente,* $V[i]$ *representa o i-ésimo elemento de* A $(1 \leq i \leq n)$*. Subvetores de* V *serão denotados por* $V[i \ldots j]$ *para quaisquer inteiros* $i \leq j$*.*

Capítulo 2

Menor Ancestral Comum

2.1 Definição

Seja $T = (V, E)$ uma árvore, e seja $v \in V$. Um ancestral de v é um vértice $u \in V$ contido no caminho de $Raiz(T)$ até v. Um ancestral comum a v_1, v_2 é, obviamente, um vértice u que é ancestral dos dois vértices v_1, v_2. Note que $Raiz(T)$ é ancestral comum a quaisquer dois vértices.

Definição 11 *Se $T = (V, E)$ é uma árvore, definimos o menor ancestral comum a dois vértices $v_1, v_2 \in V$ como o vértice u que, dentre todos os ancestrais comuns a v_1 e v_2, possui maior distância até $Raiz(T)$.*

Notação: $mac(v_1, v_2) = u$.

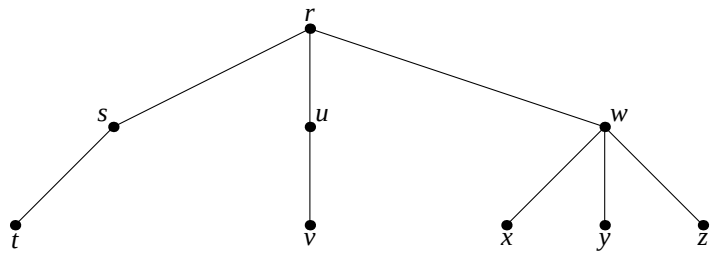

Figura 2.1: Exemplo de árvore

Na árvore acima, temos $mac(x, z) = w = mac(w, z)$

2.2 Solução eficiente

Existe uma solução para o problema do MAC que pré-processa uma árvore T de n vértices utilizando $O(n)$ unidades de tempo e espaço, e é capaz a responder a consultas u, v em tempo $O(1)$.

Vamos mostrar brevemente esta solução, ao exibirmos a relação entre o MAC e um problema aparentemente não correlato.

Considere novamente a árvore:

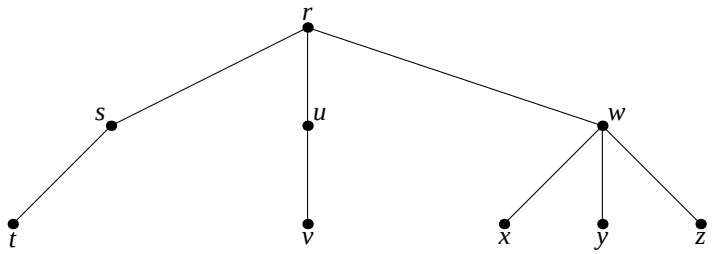

Figura 2.2: Exemplo de árvore

Chamemos de T a árvore acima. Realize uma busca em profundidade a partir de $Raiz(T)$ e considere os vetores $Busca, Prof, Desc$ e $Aban$, onde:

- *Busca[i]*: associa um instante da busca em profundidade ao vértice sendo visitado naquele instante.
- *Prof[i]*: associa um instante da busca em profundidade à profundidade do vértice $Busca[i]$.
- *Desc[v]*: associa um vértice de T ao seu instante de descoberta.
- *Aban[v]*: associa um vértice de T ao seu instante de abandono.

Temos então, neste exemplo:

$Busca[i]$	r	s	t	s	r	u	v	u	r	w	x	w	y	w	z	w	r
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Tabela 2.1: Vetor $Busca[i]$

$Prof[i]$	0	1	2	1	0	1	2	1	0	1	2	1	2	1	2	1	0
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Tabela 2.2: Vetor $Prof[i]$

$Desc[v]$	1	2	3	6	7	10	11	13	15
v	r	s	t	u	v	w	x	y	z

Tabela 2.3: Vetor $Desc[v]$

$Aban[v]$	17	4	3	8	7	16	11	13	15
v	r	s	t	u	v	w	x	y	z

Tabela 2.4: Vetor $Aban[v]$

Sejam v_1, v_2 vértices de T. Suponha, sem perda de generalidade, que temos $Desc[v_1] < Desc[v_2]$. Não é difícil ver que o Menor Ancestral Comum de dois vértices v_1, v_2 é o vértice de menor profundidade visitado no intervalo de tempo $Aban[v_1], Desc[v_2]$. Isto é, $mac(v_1, v_2)$ é o vértice visitado no instante $\operatorname{argmin} Prof[Aban[v_1] \ldots Desc[v_2]]$.

Exemplo 12 *Considere os vértices $v_1 = x$, $v_2 = z$ de nosso exemplo acima. Temos $Aban[x] = 11$, $Desc[v_2] = 13$ e ainda $\operatorname{argmin} Prof[11 \ldots 13] = 12$. Agora, $Busca[12] = w$ é o Menor Ancestral Comum de x e z.*

O vetor $Prof$ possui uma característica muito especial: seus elementos adjacentes variam sempre de ± 1. Chamaremos um tal vetor de *vetor contínuo*. Se tivermos um algoritmo que pré-processa um vetor contínuo eficientemente e, a partir de então possa responder consultas (i, j) informando $\operatorname{argmin} Prof[i \ldots j]$, teremos uma solução de igual eficiência assintótica para o MAC.

2.3 Uma solução eficientemente para o MVC

Vimos que o problema do MAC pode ser reduzido ao problema do MVC (Mínimo de um Vetor Contínuo), então esboçaremos agora uma solução eficiente do MVC.

Considere um vetor contínuo $V[1 \dots n]$ particionado em b "blocos", com $b \simeq \log n$. Ao recebermos uma consulta (i, j), iremos consultar três mínimos parciais (que foram calculados durante o pré-processamento) e retornar o mínimo entre eles:

1. O mínimo interno a um bloco inicial,
2. O mínimo de uniões de blocos inteiros e
3. O mínimo interno a um bloco final

Para calcular o segundo mínimo, calculamos explicitamente o mínimo de todas as uniões possíveis de blocos inteiros, utilizando tempo e espaço $O(n)$.

O primeiro e terceiro mínimos parciais também podem ser calculados com a mesma complexidade.[1].Chame de *normalizado* um vetor contínuo cujo primeiro elemento seja zero.

Note que existem $O(2^b) = O(n)$ possíveis blocos normalizados, pois o próximo elemento é sempre uma escolha entre aumentar o elemento anterior de 1 ou diminuir o elemento anterior de 1.

Então pré-calcularemos as posições dos mínimos de cada bloco normalizado possível e guardamos estas posições para responder às futuras consultas. O leitor interessado pode conhecer mais a respeito em [APdL03].

[1] Esta técnica é conhecida como "Quatro russos".

Capítulo 3

Resumo geral

Neste momento temos as definições necessárias para sintetizar o restante do texto.

Dada uma árvore T, qual(is) a(s) finalidade(s) de calcularmos $mac(v_1, v_2)$ de maneira eficiente? Não pretendemos responder integralmente esta pergunta (algo aparentemente impraticável): todo o texto subseqüente é um esforço direcionado à obtenção de uma resposta parcial desta questão.

Mais especificamente, nos ateremos a exibir três das finalidades supracitadas:

1. Calcular eficientemente o mínimo de qualquer subvetor de um vetor previamente dado.

2. Dado um conjunto de palavras $\{d_1, \ldots, d_k\}$, responder eficientemente em quais documentos d_i uma palavra p qualquer ocorre.

3. Dadas palavras s, t, determinar o comprimento de sua subseqüência comum máxima em tempo $O(|s| + |t| + d^2)$, onde d é uma grandeza que mede as diferenças entre s e t.

O próximo capítulo aborda a primeira finalidade exposta acima; seguido por um capítulo destinado à definição de uma estrutura de dados (A^+ árvore de reconhecimento), visando a obtenção dos pré-requisitos indispensáveis para discorrermos acerca do segundo e terceiro itens da enumeração. Por fim, segue a parte subjetiva desta monografia.

Capítulo 4

Mínimo de um vetor e árvores cartesianas

4.1 Descrição do problema

Seja $A[1 \ldots n]$ um vetor de inteiros. Queremos algoritmos eficientes para descobrir qual é o menor elemento do subvetor $A[i \ldots j]$, onde os inteiros $i \leq j$ constituem uma consulta.

Se quisermos fazer várias consultas em um certo vetor A, seria desejável pré-processá-lo e então ser capaz de responder a consultas em tempo constante. Utilizando uma solução eficiente para o MAC conseguiremos pré-processar A em tempo e espaço lineares no seu número de elementos; a partir daí seremos capazes [BM00] de responder a consultas em tempo $O(1)$.

Definição 13 *Uma árvore cartesiana T associada a um vetor $A[1 \ldots n]$ é uma árvore tal que, cada vértice u armazena um elemento do vetor $A[1 \ldots n]$ e sua respectiva posição; isto é, $u = (A[i], i)$ para algum i com $1 \leq i \leq n$. Uma árvore cartesiana T é então recursivamente definida por:*

- *A raiz de T armazena o mínimo do vetor $A[1 \ldots n]$ e uma posição onde este mínimo ocorre. Chame de k esta posição. (lembramos que o mínimo pode ocorrer em várias posições).*
- *A subárvore enraizada à esquerda da raiz de T é a árvore cartesiana do subvetor $A[1 \ldots k-1]$*
- *A subárvore enraizada à direita da raiz de T é a árvore cartesiana do subvetor $A[k+1 \ldots n]$*

Observação: é claro que, na definição, fica subentendido que se os subvetores $A[1 \ldots k-1]$ ou $A[k+1 \ldots n]$ forem vazios, então não há subárvore enraizada naquela direção (esquerda ou direita).

Exemplo 14 *Considere o vetor A, onde:*

$A[i]$	11	18	30	20	30	40	19
i	1	2	3	4	5	6	7

Tabela 4.1: Vetor A

Sua árvore cartesiana é:

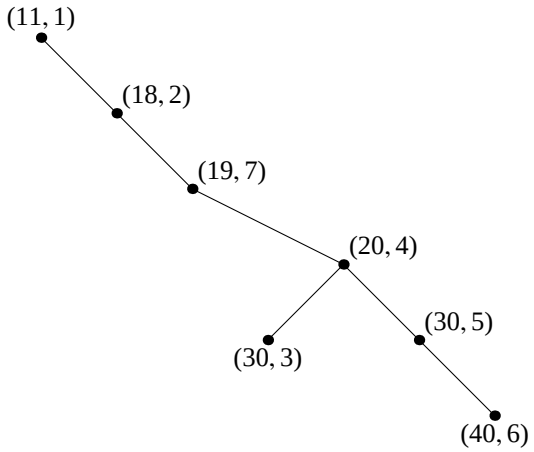

Figura 4.1: Árvore cartesiana de A

Resolver uma instância do problema do MAC em uma árvore cartesiana é exatamente o que resolverá o problema do mínimo de um vetor. Antes de mostrar isso, mostraremos que uma árvore cartesiana de um vetor pode ser construída em tempo e espaço lineares no seu número de elementos.

Proposição 15 *Dado um vetor $A[1 \ldots n]$, existe um algoritmo que constrói uma árvore cartesiana de A em $O(n)$ unidades de tempo e espaço.*

Prova. Chamemos de T a árvore cartesiana a ser construída. Primeiramente construiremos a árvore T_1, que será a árvore cartesiana (trivial) de $A[1 \ldots 1]$. Após isso, construiremos cada T_{i+1} a partir de T_i. Chame o novo vértice

$(A[i+1], i+1)$ de u. Chame de "caminho mais à direita" de T_i a seqüência de vértices contidos no caminho[1] de $(A[i], i)$ até $Raiz(T_i)$. Esta seqüência de vértices é tal que apenas filhos à direita são considerados.

Note que u pertencerá ao caminho mais à direita da árvore de T_{i+1}.

A construção se dá da seguinte forma: Compare $A[i+1]$ com os valores armazenados nos vértices do caminho mais à direita até achar um vértice $v = (A[k], k)$ tal que $A[k] < A[i+1]$. Agora, temos dois casos:

1. Se tal vértice v for achado, o vértice u será o novo filho à direita de v e a subárvore que estava à direita de v será agora subárvore à esquerda de u.

2. Caso v não exista, (isto é, $A[i+1]$ é menor do que todos os valores dos vértices do ramo à direita de T_i), o vértice u será a raiz de T_{i+1}, e toda T_i será subárvore à esquerda de u.

Sendo a construção desta forma, cada comparação acarreta uma adição ou uma remoção de vértice do caminho mais à direita. Como cada vértice só é adicionado ao ramo mais à direita uma vez e é removido do ramo mais a direita no máximo uma vez, o número de comparações do algoritmo é $O(n)$. Além disso, $T = T_n$ possui n vértices, portanto pode ser representada em $O(n)$ unidades de espaço. ∎

[1] Isto não é um caminho segundo a definição usual de árvore, e sim um caminho na "contra-mão".

Exemplificamos a construção de T_7 em função de T_6 para o vetor A considerado anteriormente:

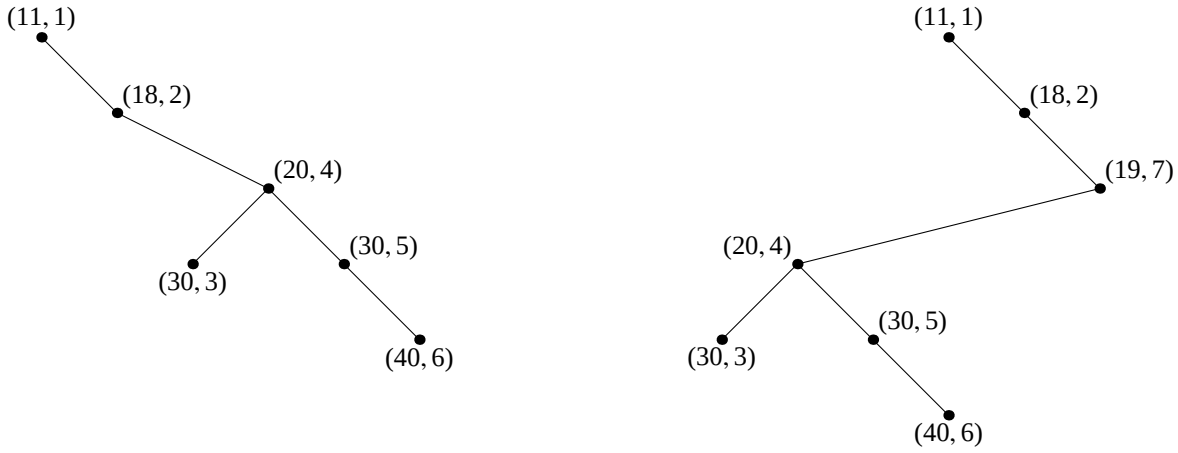

Figura 4.2: Último passo da construção de uma árvore cartesiana de A

A árvore da esquerda é T_6, uma árvore cartesiana do subvetor $A[1 \ldots 6]$. A árvore à direita é T_7, uma árvore cartesiana de $A[1 \ldots 7] = A$.

Proposição 16 *Se $A[1 \ldots n]$ é um vetor e T é uma árvore cartesiana de A, então o mínimo do subvetor $A[i \ldots j]$ é o valor armazenado no vértice $MAC_T(u, v)$ onde $u = (A[i], i)$ e $v = (A[j], j)$.*

Prova. Sejam i, j inteiros com $i < j$. Seja $w = (A[k], k) = MAC_T(u, v)$, onde u, v são como no enunciado da proposição.

Inicialmente, precisamos mostrar que u está na subárvore à esquerda de w e que v está na subárvore à direita de w.

Considere T_w (a subárvore de T enraizada em w). É claro que $u, v \in T_w$ (pois w é ancestral de u e v). Agora, observamos que u, v não podem ambos estar na subárvore à esquerda (ou direita) de w (caso contrário, haveria um ancestral comum a u e v mais distante da raiz do que w). Então u e v estão em subárvores diferentes. Agora, é fácil notar que se u pertencesse à subárvore à direita de w e se v pertencesse à subárvore à esquerda de w, teríamos uma contradição com a hipótese $i < j$. Assim, está provada a consideração inicial.

Pela definição de árvore cartesiana, $A[k]$ é o mínimo de algum subvetor maximal $A[\alpha \ldots \beta]$, isto é, $A[k] = \min A[\alpha \ldots \beta]$ onde α é o menor possível e β é o maior possível.

Pela consideração inicial, o vértice u encontra-se na subárvore à esquerda de w, então $A[i] = \min A[\gamma \ldots \delta]$ para algum $\gamma \geq \alpha$ e algum $\delta \leq k - 1$. Portanto, $\alpha \leq i \leq k - 1$.

Novamente pela consideração inicial, o vértice v encontra-se na subárvore à direita de w, então $A[j] = \min A[\tau \ldots \epsilon]$ para algum $\tau \geq k + 1$ e algum $\epsilon \leq \beta$. Portanto, $k + 1 \leq j \leq \beta$.

Por fim, $\min A[\alpha \ldots \beta] = \min A[i \ldots j]$ pois $\alpha \leq i \leq k \leq j \leq \beta$, e a proposição está provada. ∎

Temos então um algoritmo que pré-processa um vetor $A[1 \ldots n]$ em $O(n)$ unidades de tempo, e um algoritmo que responde consultas (i, j) informando o valor $\min A[i \ldots j]$ em tempo constante. É fácil ver que, se quisermos obter uma posição onde o mínimo do subvetor $A[i \ldots j]$ ocorre (isto é, queremos um $\operatorname{argmin} A[i \ldots j]$), basta modificar trivialmente o retorno do algoritmo de consultas, uma vez que o vértice de uma árvore cartesiana armazena um valor e sua respectiva posição.

Determinar a posição do mínimo de um vetor de maneira eficiente será muito importante para o algoritmo da listagem de documentos.

Capítulo 5

A^+ Árvore de Reconhecimento

5.1 A^+ Árvores

Para todas as seguintes definições, seja A um alfabeto.

Definição 17 *Uma A^+ árvore é uma tripla $T = (V, E, \lambda)$ onde (V, E) é uma árvore e $\lambda : E \to A^+$ é uma rotulação nas arestas. Além disso, exigimos que duas arestas distintas com mesmo vértice inicial possuam rótulos cujas primeiras letras são distintas.*

Definição 18 *A rotulação λ de uma A^+ árvore é naturalmente estendida para os vértices: $\lambda(v)$ é a concatenação dos rótulos das arestas que formam o caminho de $Raiz(T)$ até v.*[1]

É fácil mostrar, utilizando as duas definições acima, que os rótulos dos vértices de uma A^+ árvore são todos distintos.

Definição 19 *Seja W um conjunto de palavras. Diremos que uma A^+ árvore (T, E, λ) reconhece W se existir um subconjunto de vértices $F \subseteq V$ tal que $\lambda(F) = \{\lambda(f) : f \in F\} = W$. Neste caso, chamamos cada vértice $f \in F$ de vértice final.*

Uma A^+ árvore T que reconhece W desempenha um papel análogo ao de um autômato finito determinístico que reconhece W. As arestas de T fazem

[1] Além disso, $\lambda(Raiz(T)) = 1$, por definição, para qualquer A^+ árvore T.

papel análogo às transições de estado de um autômato[2]. Nos dois casos, os vértices finais coincidem e $Raiz(T)$ faz o papel de "estado inicial" de um autômato. Além disso, arestas e vértices artificiais eventualmente precisam ser adicionados a uma A^+ árvore para obtermos um autômato.

5.2 A^+ Árvore de Reconhecimento

O conceito de A^+ árvore de reconhecimento é motivado pela seguinte indagação natural: qual é a "menor"[3] A^+ árvore que reconhece um fixado conjunto W de palavras?

Definição 20 *Uma palavra $x \in A$ é dita prefixo de um conjunto de palavras W se x for prefixo de alguma palavra $w \in W$.*

Definição 21 *Seja W um conjunto de palavras. Uma bifurcação de W é uma palavra $x \in A^+$ tal que x é prefixo de W e existem letras distintas $a, b \in A$ tais que xa e xb são prefixos de W. Denotamos o conjunto das bifurações de W por $Bifurc(W)$.*

Definição 22 *A A^+ árvore de reconhecimento de um conjunto de palavras $W \subseteq A^+$ é a a A^+ árvore (V, E, λ) onde*

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

É possível provar que, a A^+ árvore de reconhecimento de $W \subseteq A^+$ definida acima de fato reconhece W conforme nossa definição [APdL03].

5.3 Um caso particular: A Árvore dos Sufixos

Um caso particular importante a se considerar é a A^+ árvore de reconhecimento de $W = Suf(x)$, onde $Suf(x)$ é o conjunto dos sufixos da palavra x.

[2] É necessário pensar em uma adaptação: as transições do autômato devem estar rotuladas com símbolos, e não palavras sobre A.

[3] Não entraremos no mérito de medir o tamanho de uma A^+ árvore.

Neste caso, chamamos esta A^+ árvore de reconhecimento de W simplesmente por Árvore dos Sufixos.

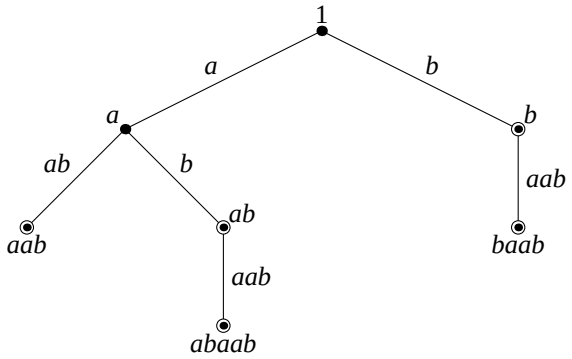

Figura 5.1: Árvore dos Sufixos de $abaab$. Os vértices finais estão destacados com dois círculos.

Será útil para nós lidarmos com Árvores dos Sufixos cujas arestas que partam de um vértice v tenham rótulos em ordem lexicográfica. Convencionaremos que as arestas que partem de qualquer $v \in V$ da esquerda para a direita estão rotuladas com palavras em ordem lexicográfica crescente.

5.4 A Árvore dos Sufixos de um conjunto de palavras

A generalização do conceito de Árvore dos Sufixos de uma palavra para o conceito de Árvore dos Sufixos de um conjunto de palavras é imediata: basta considerar a A^+ árvore de reconhecimento de $Suf(W)$, onde W é um conjunto de palavras.

Comumente nos referiremos a essa generalização do conceito de Árvore dos Sufixos simplesmente por "Árvore dos Sufixos Generalizada".

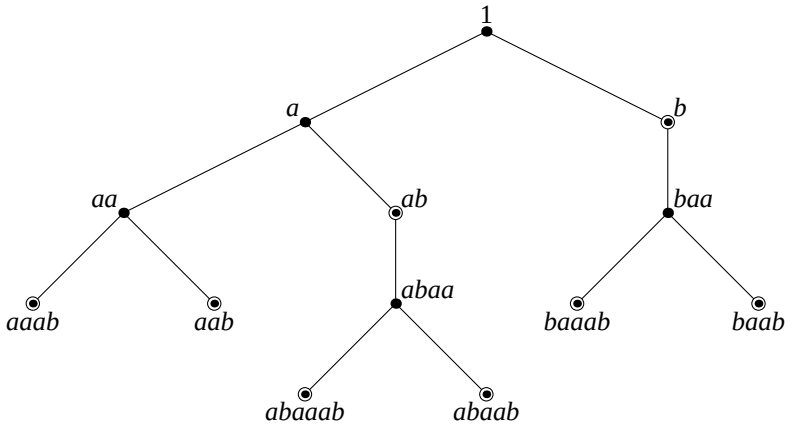

Figura 5.2: Árvore dos Sufixos de $\{abaab, abaaab\}$. A rotulação das arestas foi omitida.

5.5 Uma construção eficiente de Árvores dos Sufixos

Pode ser demonstrado [P.73] que a Árvore dos Sufixos Generalizada de um conjunto W de palavras pode ser construída com complexidade $O(n)$ de tempo e espaço, onde n é a soma dos comprimentos das palavras em W.

Capítulo 6

Listagem de documentos

6.1 Descrição do problema

Seja $B = \{d_1, \ldots, d_k\}$ um conjunto de palavras que serão chamadas de "documentos"; comumente, chama-se o conjunto B de "biblioteca". Além disso, definimos o tamanho da biblioteca como $n \stackrel{\text{def}}{=} \sum_i |d_i|$.

No problema da listagem de documentos, o conjunto B é dado e permite-se um pré-processamento. As consultas consistem em uma palavra p e o objetivo é devolver o subconjunto de B dos documentos que contém uma ou mais ocorrências de p. Mais formalmente, um documento d_i contém uma ocorrência de p se e somente se p é um fator de d_i.

A Árvore dos Sufixos Generalizada será a estrutura adequada para armazenarmos todos os sufixos da biblioteca B.

6.2 Um algoritmo eficiente

Seja B uma biblioteca dada. Considere T, a Árvore dos Sufixos Generalizada de B. Chame de $f_1, f_2, \ldots, f_j$ os vértices finais de T descobertos nesta ordem após uma busca em profundidade em T.

Em geral, pode haver um vértice f_i cujo rótulo $\lambda(f_i)$ seja sufixo de dois (ou mais) documentos d_x, d_y. Vamos exemplificar esse fenômeno no exemplo seguinte. Suponha que temos $B = \{abaab, abaaab\}$.

No caso do exemplo acima, o vértice f_2 (rotulado com aab), possui rótulo pertencente a dois documentos (aab é sufixo tanto de $abaab$ quanto de $abaaab$). Por conta disso, suporemos que a árvore T é construida de forma

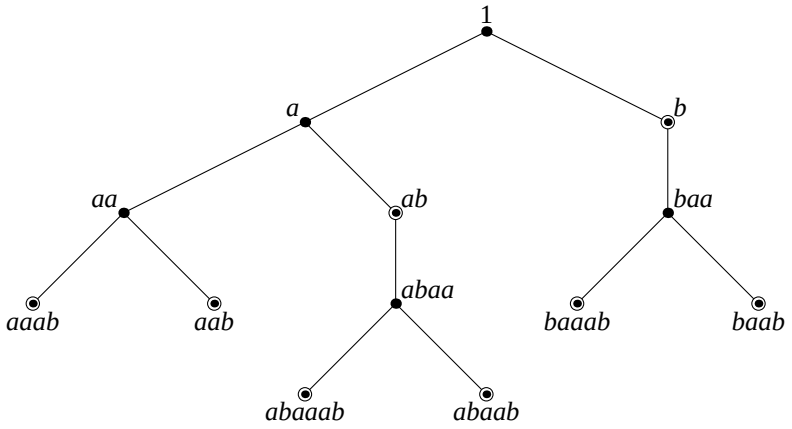

Figura 6.1: Árvore dos Sufixos de $\{abaab, abaaab\}$

a conter [1], em cada vértice v, uma lista ligada que armazena os índices dos documentos dos quais $\lambda(v)$ é sufixo. Por simplicidade, suporemos também que os índices dos documentos que figuram nestas listas ligadas estão em ordem crescente[2].

Novamente, considerando o exemplo, o vértice f_2 conteria uma lista ligada de dois elementos: 1 e 2. Já o vértice f_1, por ser rotulado com $aaab$, conteria uma lista ligada de um elemento: 2.

Considere todos os valores das listas ligadas dos vértices $f_1, f_2, \ldots, f_j$ (nesta ordem) e defina o vetor D como o vetor que armazena estes valores. Ainda no exemplo acima, teríamos:

$D[i]$	2	1	2	1	2	2	2	1	2	2	1
i	1	2	3	4	5	6	7	8	9	10	11

Tabela 6.1: Vetor D associado à Árvore dos Sufixos do exemplo

É importante notar que D possui exatamente n elementos (precisamente o número de sufixos da biblioteca, contando repetições, isto é, sufixos que ocorrem em mais de um documento).

[1] Esta suposição não compromete a complexidade da construção eficiente de uma Árvore dos Sufixos.

[2] A ordem dos índices contidos em uma certa lista ligada não altera o funcionamento do algoritmo; a suposição é feita apenas para facilitar a exemplificação.

Definição 23 *Seja p uma palavra. O lugar de p, denotado u_p, é o vértice da Árvore de Sufixos Generalizada tal que $\lambda(u_p)$ possui p como prefixo e tem $|\lambda(u_p)|$ mínimo.*

Note que se p não ocorrer em nenhum documento, então u_p não existe. É possível mostrar que [P.73] a Árvore de Sufixos Generalizada de uma biblioteca pode ser construída em $O(n)$ unidades de tempo e espaço e, dada qualquer palavra p, em $O(|p|)$ unidades de tempo podemos determinar seu lugar u_p ou afirmar que u_p não existe.

Agora descreveremos brevemente uma primeira estratégia para um algoritmo eficiente para a Listagem de Documentos.

O pré-processamento constitui-se em:

1. Construir T, a Árvore dos Sufixos Generalizada de B.
2. Fazer uma busca em profundidade em T, construindo o vetor D.

Antes de mostrarmos uma primeira maneira de responder consultas, fazemos as seguintes observações:

1. Dado p, considere seu lugar u_p. Se u_p não existir, a resposta à consulta é vazia, isto é, p não ocorre em nenhum documento de B.
2. Defina f_s como o primeiro vértice final descoberto à esquerda de u_p e f_b como o último vértice final descoberto à direita de u_p. (Nosso pré-processamento encarregará-se de armazenar as referências para f_s e f_b em cada vértice de T, durante a busca em profundidade).
3. Agora, $\lambda(f_s)$ possui $\lambda(u_p)$ como prefixo (e portanto p) e é lexicograficamente o menor sufixo da biblioteca com essa propriedade. Analogamente, $\lambda(f_b)$ é lexicograficamente o maior sufixo da biblioteca dentre aqueles que começam com $\lambda(u_p)$.

Portanto, um sufixo na biblioteca começa com p se e somente se algum dos vértices finais $f_s, \ldots, f_b$ possui p como prefixo de seu rótulo.

Problema 24 *(Consulta de cores) Dado um vetor $A[1 \ldots n]$ de "cores" para ser pré-processado, as consultas consistem em inteiros (i, j) com $1 \leq i \leq j \leq n$; a resposta à consulta é o conjunto de "cores" distintas que ocorrem em $A[i \ldots j]$.*

O problema da listagem de documentos pode ser reduzido ao problema acima da seguinte maneira: Considere o vetor D definido anteriormente, e seja o índice de cada documento sua cor. Então a consulta com palavra p resume-se a encontrar as cores distintas em $D[s \dots b]$ (aqui, s e b são os índices dos vértices finais definidos na segunda observação acima).

O melhor algoritmo para o problema da consulta de cores é capaz de resolvê-lo em tempo $O(\log n + numdocs)$ [eML93] (aqui, $numdocs$ é o número de documentos onde p ocorre). No entanto, isso não será suficiente para o desenvolvimento do nosso algoritmo: procuramos uma solução que determine quais são os documentos que contém p em $O(numdocs)$ (após o lugar de p já ter sido encontrado).

A idéia básica para resolver essa questão é a seguinte: não resolveremos a consulta de cores no vetor D. Ao invés disso, construiremos apropriadamente um vetor C juntamente à construção do vetor D, e resolveremos o problema do "mínimo de um vetor". Defina o vetor C da seguinte maneira: $C[i] = j$ se e somente se $j < i$ e $D[i] = D[j]$ e j é o maior inteiro com essa propriedade. Se tal j não existir, definimos $C[i] = -1$.

$C[i]$	-1	-1	1	2	3	5	6	4	7	9	8
i	1	2	3	4	5	6	7	8	9	10	11

Tabela 6.2: Vetor C associado à Árvore dos Sufixos do exemplo

Lema 25 *O documento i contém p se e somente se existe exatamente um $k \in [s, b]$ tal que $D[k] = i$ e $C[k] < s$.*

Prova. Omitida. O leitor pode conferi-la em [Mut]. ∎

Utilizando o lema acima, resolveremos o problema da listagem de documentos mais eficientemente da seguinte maneira [Mut]: encontramos o índice do mínimo $\alpha = \operatorname{argmin} C[s \dots b]$. Se $\alpha \geq s$, então a resposta à consulta é vazia (ou seja, p não ocorre em nenhum documento d_i) . Se $\alpha < s$, então imprimimos $D[\alpha]$ e repetimos esse procedimento nos dois subvetores: encontraremos o índice do mínimo em $C[s \dots \alpha - 1]$ e $C[\alpha + 1 \dots b]$.

Resumiremos agora então a estratégia definitiva do algoritmo para a listagem de documentos:

O algoritmo de pré-processamento recebe uma biblioteca B e consiste em:

1. Construir T, a Árvore dos Sufixos Generalizada de B.

2. Fazer uma busca em profundidade em T, enumerando os vértices finais de T (na ordem em que são descobertos), salvando em cada vértice referências para f_s e f_b, e construindo os vetores C e D.

3. Pré-processar o vetor C com um algoritmo eficiente que resolva o problema do mínimo de um vetor.

O algoritmo que responde as consultas recebe a palavra p e consiste em:

1. Encontrar u_p, o lugar de p.

2. Em u_p, ler a referência para f_s e f_b.

3. Recursivamente repetir a seguinte rotina: Encontre a posição do mínimo de $C[s \ldots b]$, chame-a de k. Se $C[k] \geq s$, então pare. Senão, imprima $D[k]$ e repita a rotina nos subvetores $C[s \ldots k-1]$ e $C[k+1 \ldots b]$.

Já foi mencionado que a Árvore dos Sufixos Generalizada de B pode ser construída usando tempo e espaço $O(n)$ (lembramos que $n \stackrel{\text{def}}{=} \sum_i |d_i|$ é o tamanho da biblioteca). A busca em profundidade citada e a construção dos vetores C e D também são feitas com a mesma complexidade. Já vimos que o pré-processamento do vetor C visando consultas de mínimos de subvetores de C é também linear em n (os vetores C e D possuem precisamente n elementos). Portanto, o pré-processamento deste algoritmo para a listagem de documentos é $O(n)$.

A resposta às consultas encontra o lugar de p, que pode ser feito em tempo $O(|p|)$. Depois disso, fazemos algumas consultas a mínimos de subvetores do vetor C, gastando tempo constante para cada consulta. Mais precisamente, fazemos $numdocs$ consultas, já que para cada documento d_i onde p ocorre há apenas um $k \in [s, b]$ com $C[k] < s$ e $D[k] = i$. Dessa maneira, o algoritmo que responde a consultas é $O(|p| + numdocs)$.

Capítulo 7

LCS

7.1 Descrição do problema

Seja $s = x_1x_2 \ldots x_n$ uma palavra. Uma subseqüência de s é uma palavra $u = u_1u_2 \ldots u_k$ tal que existem inteiros $\alpha_1 < \alpha_2 < \ldots < \alpha_k$ que satisfazem $u_i = s_{\alpha_i}$ para todo $1 \leq i \leq k$.

Uma subseqüencia comum a s e t é uma palavra u que é subseqüência de s e também de t.

Por definição, a palavra vazia λ é subseqüência comum a quaisquer duas palavras. É fácil ver que o conjunto de subseqüências comuns a duas palavras é finito (devido à finitude do comprimento das palavras). Desta forma, dadas duas palavras, sempre existe uma subseqüência comum de comprimento máximo (não necessariamente única).

Exemplo 26 *A subseqüência comum de comprimento máximo não é única. Considere $s = abca$, $t = acba$. Por força bruta, é fácil ver que ac e ab são as subseqüências comuns mais compridas.*

Denotaremos uma subseqüência comum a s e t mais comprida simplesmente por $lcs(s, t)$.

7.2 Motivação

O problema de encontrar $lcs(s, t)$ é um problema de otimização que surge naturalmente em muitas aplicações práticas, como na comparação de seqüências

genéticas e na determinação de similaridades entre dois arquivos texto, por exemplo.

7.3 Algoritmo O(nm) (n = |s| , m = |t|)

Apresentaremos um algoritmo que usa programação dinâmica para a solução do LCS. Defina $lcs\_p(i,j)$ como o comprimento da subseqüência comum mais comprida de $s[1\ldots i]$ e $t[1\ldots j]$. Não é difícil ver que $lcs\_p(i,j)$ satisfaz a seguinte recorrência:

$$lcs\_p(i,j) = \begin{cases} lcs\_p(i-1,j-1)+1 & \text{, se } s[i]=t[j] \\ \max\{lcs\_p(i-1,j),\ lcs\_p(i,j-1)\} & \text{, caso contrário} \end{cases}$$

para todo $1 \leq i \leq |s|$ e todo $1 \leq j \leq |t|$.

$lcs\_p(i,0) = lcs\_p(0,j) = 0$
para todo $0 \leq i \leq |s|$ e todo $0 \leq j \leq |t|$.

O seguinte algoritmo calcula, dados s e t, o comprimento $|lcs(s,t)|$. A idéia do algoritmo é calcular $lcs\_p(i,j)$ primeiramente para valores pequenos de i e j, até calcular $lcs\_p(|s|,|t|) = |lcs(s,t)|$ (isto é, o algoritmo implementa uma solução bottom-up para o problema).

Algorithm 1 Algoritmo dinâmico para o LCS

LCS-DYN($s[1...n], t[1...m]$)

```
1  para i de 0 até n faça
2        C[i,0] ← 0
3  para j de 0 até m faça
4        C[0,j] ← 0
5  para i de 0 até n faça
6        para j de 0 até m faça
7              se s[i] = t[j] então
8                    C[i,j] ← C[i−1,j−1] + 1
9              senão C[i,j] ← max{C[i−1,j], C[i,j−1]}
10 devolva C[n,m]
```

É fácil modificar o algoritmo de forma a obter de fato a palavra $lcs(s,t)$, e não somente o comprimento $|lcs(s,t)|$. Basta alocarmos, adicionalmente,

uma matriz $B[i,j]$ de mesmas dimensões que $C[i,j]$, e "registrarmos a origem" dos valores $C[i,j]$.

7.4 Preliminares para um segundo algoritmo para o LCS

Sejam s,t duas palavras sobre Σ, sejam $n = |s|$, $m = |t|$. Antes de começarmos uma série de definições para o entendimento de um segundo algoritmo para o LCS, iremos definir mais parâmetros do problema. Reservaremos a letra l para o comprimento do $lcs(s,t)$. Definimos também a seguinte grandeza: $d \stackrel{\text{def}}{=} n+m-2l$. Para esta definição não parecer totalmente arbitrária, motivaremos esta atitude informalmente no parágrafo seguinte.

Podemos pensar no número mínimo de operações necessárias para transformar a em b. Podemos "remover" de a os $n-l$ caracteres que não aparecem em $lcs(s,t)$. Em seguida, "adicionamos" os $m-l$ caracteres presentes em b que não estão em $lcs(s,t)$. Temos então $n-l+m-l = n+m-2l$ "operações", e está assim de certa maneira motivada a definição de d.

O leitor pode interpretar l da seguinte maneira: l é maior quanto maior for a "semelhança" entre a e b. De maneira oposta, d é maior quanto maior for a "diferença" entre a e b.

Introduziremos um formalismo bastante intuitivo para a compreensão de um segundo algoritmo para o cálculo do LCS entre duas palavras [Mye]. Assim como no algoritmo dinâmico, a princípio nos preocuparemos em descrever um algoritmo que encontre o comprimento $|lcs(s,t)|$.

7.5 O algoritmo O((n+m)d) de Myers

Definição 27 *O grafo de edição das palavras s,t é um grafo $\overrightarrow{G} = (V,E)$ tal que:*

- *Para cada par de inteiros x,y, $0 \leq x \leq n$ e $0 \leq y \leq m$, existe um vértice $v \in V$. É útil pensarmos no grafo como um malha discreta (ou "grid", em inglês).*
- *Para cada par de inteiros x,y , $0 \leq x \leq n-1$ e $0 \leq y \leq m$, existe uma aresta "horizontal" cujo início é x,y e cujo término é $x+1,y$.*

- *Para cada par de inteiros* x, y , $0 \leq x \leq n$ *e* $0 \leq y \leq m-1$, *existe uma aresta "vertical" cujo início é* x, y *e cujo término é* $x, y+1$.
- *Para cada par de inteiros* x, y , $0 \leq x \leq n-1$ *e* $0 \leq y \leq m-1$, *se tivermos* $a[x] = b[y]$, *então existe uma aresta "diagonal" cujo início é* $x-1, y-1$ *e cujo término é* x, y.

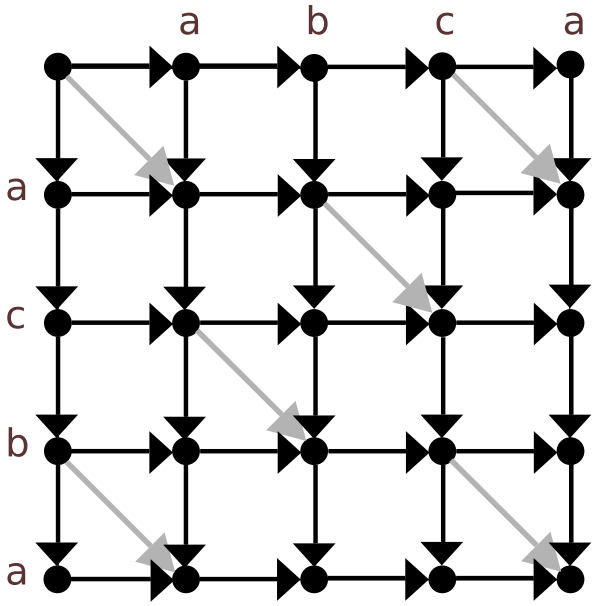

Figura 7.1: Grafo de edição de $abca, acba$. As arestas diagonais estão em cinza apenas para haver maior destaque.

Definição 28 *Chamamos de traçado de tamanho* k *um conjunto de arestas diagonais* $\{e_1, \ldots, e_k\}$ *de um grafo de edição* $\overrightarrow{G}$ *tal que* $e_1, \ldots, e_k$ *estão contidas em algum caminho de* $\overrightarrow{G}$.

Não é difícil ver que a existência de uma subseqüência comum a s e t de comprimento k implica na existência de um traçado de comprimento k no grafo de edição de s, t.

A estratégia do nosso algoritmo será encontrar um traçado de tamanho máximo no grafo de edição de duas palavras. Na realidade, o grafo não será construído explicitamente, o conceito de grafo de edição apenas auxilia o entendimento do algoritmo.

O leitor pode facilmente constatar que um traçado sempre está contido em algum caminho que inicia em $(0,0)$ e termina em (n,m). Então nosso algoritmo procurará por um caminho que inicia em $(0,0)$ e termina em (n,m) com o maior número de arestas diagonais.

Proposição 29 *Um caminho de $(0,0)$ a (n,m) com D arestas não diagonais possui $c = (n+m-D)/2$ arestas diagonais, e portanto determina uma subseqüência comum de comprimento c.*

Prova. Seja p o caminho de $(0,0)$ a (n,m) Note que as arestas de p que partem de um vértice (x,y) necessariamente aumentam o valor $x+y$. As arestas diagonais aumentam $x+y$ duas unidades, e as arestas não diagonais aumentam $x+y$ uma unidade. Então, se D é o número de arestas não diagonais e c é o número de arestas diagonais, temos $2c+D = n+m \iff c = (n+m-D)/2$ ∎

Como na proposição acima, devido à igualdade $2c+D = n+m$, é fácil ver que o caminho de $(0,0)$ a (n,m) com maior número arestas diagonais é aquele com o menor número de arestas não diagonais.

Definição 30 *Um caminho no grafo de edição que começa em $(0,0)$ e possui D arestas não-diagonais será chamado de D-caminho.*

Definição 31 *A k-ésima diagonal de um grafo de edição é o conjunto de vértices (x,y) tais que $x-y = k$.*

Definição 32 *Um D-caminho que termina na diagonal k será chamado de "mais distante" se, dentre todos os D-caminhos que terminam na diagonal k, for aquele cujo vértice final possui maior coordenada x (e portanto, maior coordenada y também).*

Exemplo 33 *Qualquer D-caminho que termina em (n,m) é mais distante.*

Agora temos as definições necessárias para tornar a estratégia do algoritmo mais explícita: procuraremos todos os D-caminhos mais distantes em ordem crescente de D. Isto é, procuraremos todos os 0-caminhos mais distantes, depois todos os 1-caminhos mais distantes e assim por diante. O algoritmo parará assim que um D-caminho mais distante terminar no vértice (n,m).

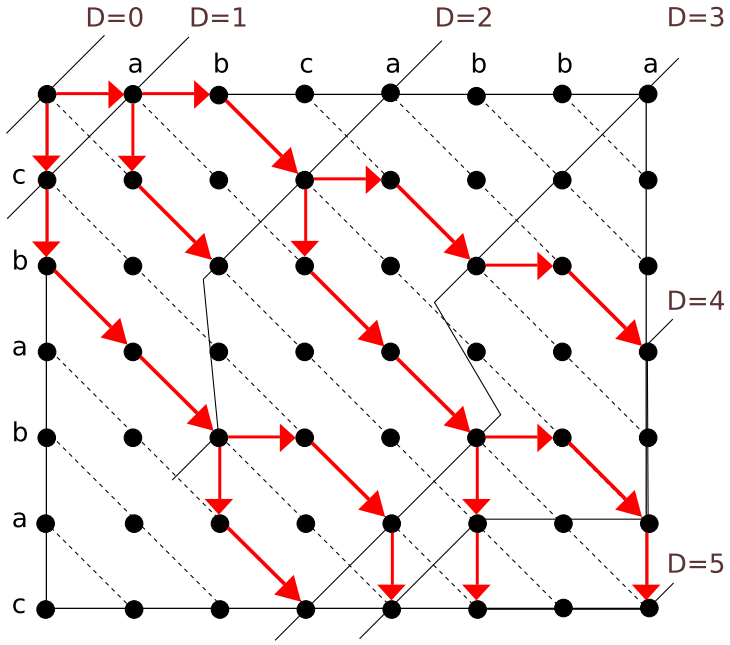

Figura 7.2: D-caminhos em um grafo de edição

Na figura acima, temos representados os limites dos D-caminhos para $0 \leq D \leq 5$. Com $D = 5$, o algoritmo pára pois um 5-caminho chegou ao vértice (n, m). Nesta imagem, só foram representadas as arestas que fazem parte de algum D-caminho percorrido pelo algoritmo.

Proposição 34 *Um D-caminho necessariamente tem seu último vértice pertencente a alguma das diagonais* $\{-D, -D+2, \ldots, D-2, D\}$.

Prova. Usaremos indução em D. Um 0-caminho é formado somente por arestas diagonais, além disso, começa na diagonal 0. Portanto, também termina na diagonal 0.

Suponha que a tese valha para D-caminhos. Isto é, um D-caminho termina na diagonal $k \in \{-D, -D+2, \ldots, D-2, D\}$. É claro que um (D+1)-caminho pode ser decomposto em um D-caminho, mais uma aresta e não diagonal seguida por algumas (ou nenhuma) arestas diagonais. Pela hipótese de indução, há duas possibilidades: e incide na diagonal $k+1$ ou na diagonal $k-1$. Como as arestas diagonais do fim da decomposição não alteram a diagonal final, temos que o (D+1)-caminho termina na di-

agonal $k \pm 1 \in \{-D \pm 1, (-D+2) \pm 1, \ \ldots \ , (D-2) \pm 1, D \pm 1\} = \{-D-1, -D+1, \ \ldots \ , D-1, D+1\}$, e está provado. ∎

Algorithm 2 Algoritmo Myers

LCS2($s[1...n], t[1...m]$)

```
1  para D de 0 até m + n faça
2      para k de −D até D passo 2 faça
3          ▷ Seja v o vértice final do D-caminho mais
4          ▷ distante que termina na diagonal k
5          se v = (n, m) então
6              ▷ Pare. O comprimento do LCS é (n + m - D) / 2
```

Procuraremos explicitar melhor o funcionamento do algoritmo acima: primeiramente diremos como construir D-caminhos mais distantes: a construção é feita de maneira "gulosa": D-caminhos mais distantes são construidos ao aumentarmos gulosamente (D-1)-caminhos mais distantes. Antes de mostrar esta construção, uma breve definição:

Definição 35 *Uma "cobra" é uma conjunto de arestas diagonais que formam um caminho.*

O seguinte lema justifica nossa construção de D-caminhos mais distantes:

Lema 36 *Um 0-caminho mais distante termina em $(x-1, x-1)$ onde $x = \min_{z \geq 1} \{a[z] \neq b[z]$ ou $z > M$ ou $z > N\}$. Se $D \geq 1$, então um D-caminho mais distante sempre pode ser decomposto de uma das seguintes formas:*

- *Um (D-1)-caminho mais distante que termina na diagonal k-1, seguido por uma aresta horizontal, seguida pela maior cobra possível.*
- *Um (D-1)-caminho mais distante que termina na diagonal k+1, seguido por uma aresta vertical, seguida pela maior cobra possível.*

Prova. Não demonstraremos este lema por acreditarmos ser de fácil constatação. O leitor pode checar [Mye]. ∎

Agora mostramos o algoritmo em toda plenitude, sem detalhes omitidos:

Algorithm 3 Algoritmo Myers

```
LCS3(s[1...n], t[1...m])
 1  V[1] ← 0
 2  para D de 0 até m + n faça
 3        para k de −D até D passo 2 faça
 4              se k = −D ou (k ≠ D e V[k − 1] < V[k + 1]) então
 5                    x ← V[k + 1]
 6              senão x ← V[k − 1] + 1
 7              y ← x − k
 8              enquanto x < n e y < M e a_{x+1} = b_{y+1} faça   ▷ percorre a maior
 9                    x ← x + 1                                   ▷ cobra possível
10                    y ← y + 1
11              V[k] ← x
12              se x ≥ N e y ≥ M então
13                    ▷ Pare. O comprimento do LCS é (n + m - D) / 2
```

Quem faz o papel de armazenar os D-caminhos mais distantes é o vetor V, que guarda a abcissa do vértice final dos D-caminhos mais distantes que terminam em cada diagonal $k = -D, -D+1, \ldots, D-1, D$. Não é necessário guardar a outra coordenada do vértice final pois y pode ser calculado através de $y = x - k$, como de fato é feito na linha 7.

7.6 Análise do consumo de tempo

Quantas iterações são feitas do laço externo (linha 2) ? Por construção, o algoritmo pára somente quando $l = (n+m-D)/2$, isto é, $D = n+m-2l = d$. Portanto, o laço externo itera $d+1$ vezes (pois D é inicialmente zero).

Agora, na k-ésima iteração do laço externo, o laço interno (linha 3) é iterado exatamente k vezes. Então o número total de iterações do laço interno é $1+2+\ldots+(d+1) = (d+1)(d+2)/2$. Todas as linhas internas aos laços executam em tempo constante, a menos do while (linhas 8,9 e 10). Então, o algoritmo gasta, a menos do bloco do while, tempo $O((d+1)(d+2)/2) = O(d^2)$.

O bloco de while (linhas 8,9,10) é executado uma vez a cada aresta diagonal presente em um D-caminho mais distante sendo construido. O algoritmo só procura caminhos nas diagonais $-d, -d+1, \ldots, d-1, d$, totalizando $2d+1$

diagonais. É claro que cada diagonal tem no máximo $\min(n, m) + 1$ vértices e portanto no máximo $\min(n, m)$ arestas. Ou seja, o algoritmo não percorre mais do que $(2d + 1)\min(n, m) = O((n + m)d)$ arestas diagonais.

Em suma: O algoritmo como um todo é $O((m+n)d)$. A análise separada do bloco do while foi feita para notar-se que o bloco do while é um "gargalo" do desempenho do algoritmo.

7.7 Melhoria da complexidade

Para melhorarmos a complexidade de pior caso, vamos mostrar como percorrer cobras em tempo $O(1)$, após feito um pré-processamento $O(n + m)$. Vamos descrever como obter essa melhoria concomitantemente à exibição de um exemplo.

Exemplo 37 *Considere novamente as palavras $s = abca$, $t = acba$.*

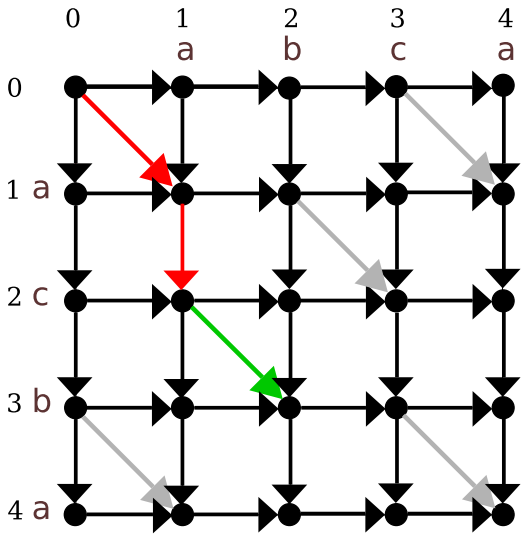

Figura 7.3: Grafo de edição de $abca, acba$

É claro que podemos identificar naturalmente os vértices de um grafo de edição com os pares $\{0, \ldots, n\} \times \{0, \ldots, m\}$. Por exemplo, quando nos referimos ao vértice $(1, 2)$ do grafo acima, estamos nos referindo ao vértice inicial da aresta verde. Vamos supor que queremos encontrar o comprimento da cobra maximal cujo vértice inicial é $(1, 2)$.

Seja (x, y) um vértice do grafo de edição das palavras s, t. O problema de encontrar a cobra maximal que começa em (x, y) é equivalente a encontrar o maior prefixo comum das subpalavras $s[x+1 \ldots n]$ e $t[y+1 \ldots m]$.

Considere $T = (V, E, \lambda)$, a Árvore dos Sufixos da palavra $s\$_1 t\$_2$, onde $\$_1$ e $\$_2$ são símbolos não pertencentes[1] ao alfabeto de s e t.

Chame de f_x o vértice final associado ao sufixo $s[x+1 \ldots n]\$_1 t\$_2$ e chame de f_y o vértice final associado ao sufixo $t[y+1 \ldots m]\$_2$. No nosso exemplo, teríamos $s[x+1 \ldots n]\$_1 t\$_2 = s[1+1 \ldots n]\$_1 t\$_2 = bca\$_1 acba\$_2$
e $t[y+1 \ldots m]\$_2 = t[2+1 \ldots m]\$_2 = ba\$_2$. Esquematicamente:

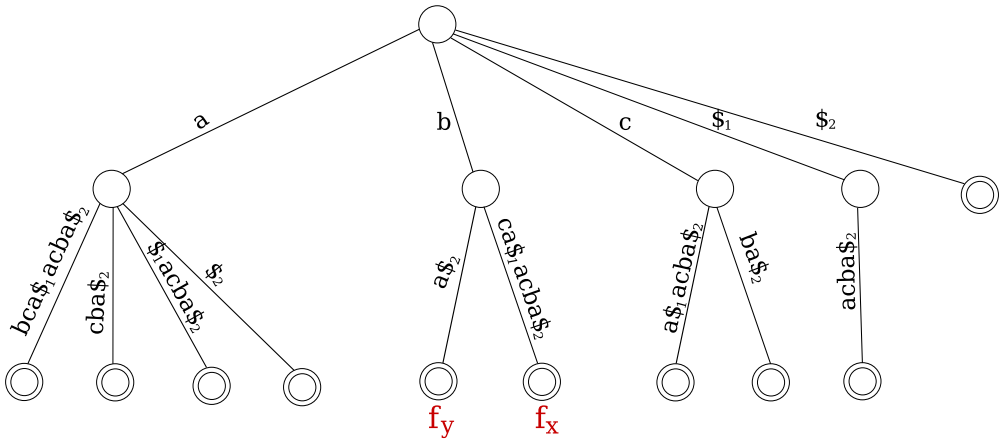

Figura 7.4: Árvore dos Sufixos de $abca\$_1 acba\$_2$

Pela definição de Árvore dos Sufixos, λ rotula arestas com palavras não vazias, portanto qualquer ancestral de f_x (respectivamente, f_y) é rotulado com um prefixo próprio de $\lambda(f_x)$ (respectivamente, $\lambda(f_y)$) (lembramos que o rótulo de um vértice v é a concatenação dos rótulos das arestas contidas no caminho de $Raiz(T)$ até v).

Então um ancestral comum a f_x e f_y é necessariamente rotulado com um prefixo comum a $s[x \ldots n]\$_1 t\$_2$ e $t[y \ldots m]\$_2$. Se considerarmos $mac(f_x, f_y)$, teremos o vértice rotulado com o maior dos prefixos comuns a essas duas subpalavras. É fácil ver (pela não ocorrência de $\$_1$ e $\$_2$ em s e t) que tal prefixo comum também é comum a s e t. Portanto, a cobra que começa em (x, y) termina no vértice $(x+k, y+k)$ onde $k = |\lambda(mac(f_x, f_y))|$. No nosso

[1] Também convencionamos, arbitrariamente, que $\$_1$ é lexicograficamente menor que $\$_2$, e isto se reflete na representação desta Árvore dos Sufixos.

exemplo, o comprimento da cobra é 1, pois o $mac(f_x, f_y)$ está rotulado com a palavra b, de comprimento 1.

7.8 Análise do algoritmo modificado

Para construir a Árvore de Sufixos de $s\$_1 t\$_2$, gastamos $O(n + m)$ unidades de tempo e espaço. A partir daí podemos obter o vértice final de uma cobra qualquer em tempo constante.

Segue facilmente da análise do algoritmo de Myers que esta modificação resulta em um algoritmo $O(m + n + d^2)$ para o cálculo do comprimento de $lcs(s, t)$.

7.9 Como obter o LCS

O algoritmo descrito encontra o comprimento de $lcs(s, t)$. Para obtermos de fato a subseqüência comum mais comprida de s e t, precisamos guardar uma cópia do vetor V a cada iteração do laço externo (linha 2). Esta modificação resulta em um algoritmo que utiliza espaço proporcional a $O(d^2)$. Omitiremos os detalhes desta versão do algoritmo.

Capítulo 8

Parte subjetiva

8.1 Escolha do tema

Inicialmente, minha intenção era escrever acerca de algum tema pertinente à área de linguagens formais e autômatos. Ao procurar o professor Alair, fui apresentado a um assunto um pouco alheio a esta área: conheci o problema do Menor Ancestral Comum e uma de suas soluções eficientes.

Conhecer esta solução me foi uma grata surpresa: em particular, fui atraído pela sua contra-intuitividade e pela revelação de uma conexão profunda do MAC com um problema aparentemente não correlato (o problema do mínimo de um vetor contínuo); estes dois fatores culminaram na execução deste trabalho.

8.2 Disciplinas mais relevantes para este trabalho

- MAC0338 - Análise de Algoritmos: O estudo do crescimento assintótico de funções, o conceito de programação dinâmica e a análise de algoritmos que resolvem problemas de otimização combinatória foram de fundamental importância para este trabalho.

- MAC0323 - Estruturas de dados: Aprender a complexidade das operações de inserção, remoção e busca em estruturas de dados diversas foi essencial para minha compreensão e consequente escrita sobre esse assunto.

- MAC0328 - Algoritmos em grafos: Muitos assuntos abordados nesta disciplina não foram usados diretamente, exceção feita à busca em profundidade, que foi de vital importância para o meu entendimento de algoritmos como o da Listagem de Documentos.

Naturalmente, inúmeras outras disciplinas do currículo contribuíram de maneira indireta para o andamento de minha iniciação científica.

8.3 Desafios e frustrações

Nesta iniciação científica, senti dificuldades para confluenciar informações oriundas de artigos distintos: em particular, o uso de notações diferentes, pelas diversas fontes de informação, foi um empecilho inicial considerável.

Apesar de não ter havido bolsa para este trabalho e, conseqüentemente, não ter havido a necessidade de reportar-me a nenhum órgão de fomento, senti dificuldade inicialmente em manter o desempenho nas disciplinas do instituto, devido à escolha de algumas matérias optativas de renomada dificuldade.

No que diz respeito a frustrações, sinto apenas não ter podido discorrer sobre a análise de caso médio do algoritmo de Myers, estabelecendo um modelo probabilístico de comparação de seqüencias aleatórias. Isto teria mostrado ao leitor que a complexidade $O((m+n)d)$ é atingida somente em instâncias de certa maneira bastante particulares.

8.4 Agradecimentos

Ao meu orientador, Alair; à minha família e a todos os professores do instituto com os quais tive contato ao longo destes anos. Por fim, agradeço também aos meus colegas e amigos, sejam de fora ou de dentro do IME.

Referências Bibliográficas

[APdL03] Imre Simon Alair Pereira do Lago. Tópicos em algoritmos sobre seqüências. *24o Colóquio Brasileiro de Matemática*, 2003.

[BM00] Michael W. Bender and Farach-Colton Martín. The LCA problem revisited. *Latin American Theoretical INformatics*, 2000.

[eML93] R. Janardan e M. Lopez. Generalized intersection searching problems. *International Journal of Computing and Geometric Applications*, pages 39–69, 1993.

[Mut] S. Muthukrishnan. Efficient algorithms for document retrieval problems. *ACM-SIAM Symposium on Discrete Algorithms*.

[Mye] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*.

[P.73] Weiner P. Linear pattern matching algorithms. *Proceedings of 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.