Algoritmos em sequências

Yan Soares Couto

Orientadora: Cristina G. Fernandes

São Paulo, 2016

Resumo

COUTO, Y. S. **Algoritmos em sequências**. Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2016.

Algoritmos em sequências são úteis para buscas de padrões e em biologia computacional. É possível descobrir a maior substring palíndroma de uma string em tempo linear. Tries são estruturas para armazenamento de string, permitindo buscas rápidas por prefixos. Pode-se usar uma trie para buscar um conjunto de strings, como padrão, em uma sequência. A construção de uma trie para todos os sufixos de uma string pode ser feita em tempo linear. Pode-se construir um autômato que reconhece todos os sufixo de uma string em espaço e tempo linear.

Palavras-chave: algoritmo, string, trie, autômato, palíndromo.

Conteúdo

Introdução

Encontrar todas as ocorrências de uma certa palavra em um texto é um problema muito importante, que vemos diariamente, por exemplo ao buscar uma palavra-chave em nossos emails recebidos ou em algum documento.

Nessa classe de problemas, de busca de ocorrências exatas, não são aceitas diferenças entre o padrão buscado e as posições de ocorrência retornadas. Outra classe de problemas é de busca de ocorrências inexatas, ou busca de ocorrências aproximadas, quando algumas diferenças são aceitas. Nesse caso é necessário adotar alguma definição de similaridade entre duas cadeias de caracteres, em geral o número mínimo de caracteres que precisam ser mudados para tornar ambas cadeias iguais. O utilitário `diff`, por exemplo, encontra o número mínimo de linhas que precisam ser mudadas para tornar dois arquivos iguais.

Outra área que se beneficia do desenvolvimento de algoritmos para tais problemas é a área da biologia computacional, pois DNA pode ser modelado como uma sequência de caracteres `A`, `T`, `C` e `G`. Assim, é possível encontrar similaridades entre trechos de DNAs diferentes, e buscar, em um conjunto enorme de trechos de DNA, algum similar ou idêntico a um trecho dado. Proteínas também podem ser modeladas como sequências de aminoácidos.

Os assuntos discutidos nesse trabalho são também muito úteis para praticantes de programação competitiva, pois os algoritmos abordados podem ser usados na resolução de vários problemas não triviais envolvendo strings, e têm implementação pequena.

Objetivos e estrutura do texto

Neste trabalho, apresentamos algoritmos e estruturas de dados relacionados a problemas de busca de ocorrências exatas. Os algoritmos clássicos para encontrar ocorrências exatas, como KMP e Boyer-Moore, não são abordados. O foco é em estruturas mais poderosas, que generalizam os algoritmos clássicos, são mais rápidas ou, quando mais complicadas, resolvem problemas mais difíceis.

Diferentemente dos livros e fontes mais comuns, a implementação é bastante discutida, com pseudocódigo usando apenas instruções presentes nas principais linguagens de programação. Porém, a teoria não é esquecida. Ao apresentar os algoritmos e suas explicações, a corretude e complexidade são provadas formalmente, em detalhes, de maneira tão completa quanto possível.

O Capítulo 1 trata de um algoritmo útil para encontrar palíndromos em uma cadeia de caracteres, e é o tema que menos se relaciona à busca de ocorrências exatas, apesar de ser bastante similar a um algoritmo que serve justamente para busca de ocorrências exatas.

O Capítulo 2 apresenta uma estrutura de dados chamada trie, com a qual é possível armazenar

um conjunto de strings de forma inteligente. Esta estrutura é usada nos dois capítulos seguintes de forma mais extensa.

O Capítulo 3 apresenta uma generalização de KMP que funciona com tries e não apenas strings, e dessa forma é possível buscar ocorrências de várias strings ao mesmo tempo.

O Capítulo 4 mostra como construir uma trie para todos os sufixos de uma string, mas sem gastar tempo ou espaço proporcional ao tamanho de todos estes. Essa estrutura é extremamente poderosa, podendo de certa forma generalizar o algoritmo do Capítulo 3.

O Capítulo 5 apresenta um autômato que identifica todos os sufixos de uma string, e é tão poderoso quanto a estrutura do Capítulo 4.

As principais fontes usadas neste trabalho foram os livros de Gusfield [Gus97] e Crochemore, Hancart e Lecroq [CHL07].

Notação e definições básicas

Uma *string* $T[1\,..\,n]$ de tamanho $|T| \coloneqq n$ é um vetor de n elementos, onde todos elementos são de um alfabeto Σ finito. Para facilitar a implementação, em geral assumimos que $\Sigma = \{0, \ldots, |\Sigma| - 1\}$, ou seja, Σ tem uma ordem e seus elementos estão "comprimidos", mas nos exemplos usamos $\Sigma = \{\mathrm{a}, \ldots, \mathrm{z}\}$.

Dizemos que $T[i\,..\,j] \coloneqq T[i]T[i+1]\ldots T[j]$ é uma *substring* de T. Se $T_1 = T_2[1\,..\,i]$, para algum $1 \leq i \leq |T_2|$, então T_1 é *prefixo* de T_2, denotado por $T_1 \sqsubseteq T_2$. Da mesma forma, se $T_1 = T_2[i\,..\,|T_2|]$ então T_1 é *sufixo* de T_2, denotado por $T_1 \sqsupseteq T_2$. Um sufixo ou prefixo é *próprio* se não é igual à string original, e nesse caso usamos $\sqsubset$ e $\sqsupset$. Ou seja, $T_1 \sqsubset T_2$ se e somente se $T_1 \sqsubseteq T_2$ e $T_1 \neq T_2$.

Dizemos que T_1 ocorre em T_2 se T_1 é uma substring de T_2. Existe uma ocorrência de T_1 em T_2 na posição i se $T_1 \sqsubseteq T_2[i\,..\,|T_2|]$, ou seja, se $T_1 = T_2[i\,..\,i + |T_1| - 1]$.

Capítulo 1

Palíndromos

1.1 Palíndromos ímpares

Seja S uma string. Dizemos que S é um *palíndromo* se $S[i] = S[|S| - i + 1]$ para todo $1 \leq i \leq |S|$. Defina $\overline{S}$ como a string $S[|S|]S[|S|-1]\cdots S[2]S[1]$, ou seja, S com caracteres de trás para frente. Alternativamente, a string S é um palíndromo se e somente se $S = \overline{S}$. Dizemos que S é um palíndromo *ímpar* se S é um palíndromo de comprimento ímpar. Nesse caso dizemos que esse palíndromo é *centrado* em $\frac{|S|+1}{2}$. A string `reviver`, por exemplo, é um palíndromo ímpar centrado no índice da letra `i`.

Vamos considerar substrings de S que são palíndromos. Vamos apresentar um algoritmo, inicialmente proposto por Manacher [Man75], que, para cada $1 \leq i \leq |S|$, determina a maior substring palíndroma ímpar de S centrada em i. Esse algoritmo consome tempo $\mathcal{O}(|S|)$. Mais precisamente, defina $\textsc{Center}(S, i, k)$ como a string $S[i-k\,..\,i+k]$, se esta estiver definida, ou seja, se $i - k \geq 1$ e $i + k \leq |S|$. Queremos criar um vetor M tal que, para cada $1 \leq i \leq |S|$, vale que $M[i]$ é o maior valor tal que $\textsc{Center}(S, i, M[i])$ é um palíndromo. Dizemos que $M[i]$ é o *alcance* de i. Note que $\textsc{Center}(S, i, 0)$ é sempre palíndromo.

Proposição 1.1.1. *Para todo $k > 0$, temos que $\textsc{Center}(S, i, k)$ é palíndromo se e somente se $\textsc{Center}(S, i, k-1)$ é palíndromo e $S[i-k] = S[i+k]$.*

Demonstração. Temos pela definição que $\textsc{Center}(S, i, k)$ é palíndromo se e somente se, para todo $1 \leq j \leq 2k+1$, vale a igualdade $S[i-k+j-1] = S[i+k-j+1]$. A string $\textsc{Center}(S, i, k-1)$ é um palíndromo quando $S[i-k+g] = S[i+k-g]$ para todo $1 \leq g \leq 2k-1$. Mas estas igualdades são equivalentes às igualdades para $j \in \{2, \ldots, 2k\}$.

Para $j = 1$, temos que $S[i-k+1-1] = S[i+k-1+1]$ e, para $j = 2k+1$, temos a igualdade $S[i-k+2k+1-1] = S[i+k-2k-1+1]$, ou seja, ambas expressam simplesmente que $S[i-k] = S[i+k]$. □

A ida da proposição implica que os números k que satisfazem que $\textsc{Center}(S, i, k)$ é um palíndromo são um prefixo de $\{0, 1, \ldots\}$. A volta mostra como descobrir se $M[i] > x$ para alguma estimativa x. Dessa forma, já é possível criar um algoritmo simples para calcular o vetor M, como no Código 1.

Código 1 Algoritmo simples para alcances

```
1: for i = 1 to |S| :
2:     x = 0
3:     while i − x > 1 and i + x < |S| and S[i − x − 1] = S[i + x + 1] :
4:         x += 1
5:     M[i] = x
```

Nesse algoritmo, a partir do zero, aumentamos a estimativa para $M[i]$, armazenada em x, enquanto temos que $x < M[i]$. As primeiras duas condições do **while** garantem que a string $\textsc{Center}(S, i, x+1)$ está bem definida. A complexidade deste código é $\Theta(\sum_{i=1}^{|S|} M[i]) = \mathcal{O}(|S|^2)$ e sua corretude segue diretamente da Proposição 1.1.1.

1.2 Uso de alcances anteriores

Vamos apresentar um algoritmo com ideia similar a esse, que também calcula os valores de M em ordem crescente de índice, e aumenta a estimativa de $M[i]$ de um em um. Porém, esse algoritmo usa os valores anteriormente calculados para começar de um valor inicial maior para x.

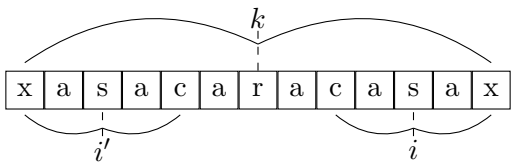


Figura 1.1: Explicação da Proposição 1.2.1.

Proposição 1.2.1. *Seja $k < i$ tal que $k + M[k] > i$. Se $i + M[2k - i] < k + M[k]$ então vale que $M[i] = M[2k - i]$. Caso contrário, temos que $M[i] \geq k + M[k] - i$.*

Demonstração. Note que, como $k + M[k] > i$, a posição i está contida na string $\textsc{Center}(S, k, M[k])$. Tome $g = k + M[k] - i$, que é a distância de i para a posição final de $\textsc{Center}(S, k, M[k])$, e $i' = k - (i - k) = 2k - i$, que é a posição i espelhada em torno do centro k. Note que $S[i'] = S[i]$, e generalizando isto, usando a definição de palíndromo, temos que $\textsc{Center}(S, i, g) = \overline{\textsc{Center}(S, i', g)}$. A Figura 1.1 mostra como isso pode ocorrer para a palavra `xasacaracasax`. Note que, nesse caso, $M[k] = 6$, $g = 2$ e temos que $\textsc{Center}(S, i, 2) = \overline{\textsc{Center}(S, i', 2)} = \texttt{casax}$.

Note que a string P é um palíndromo se e somente se $\overline{P}$ é um palíndromo. Logo, se $M[i'] < g$, como $M[i']$ é o maior palíndromo centrado em i', temos pela Proposição 1.1.1 que $S[i' - M[i'] - 1] \neq S[i' + M[i'] + 1]$, e portanto $S[i + M[i'] + 1] \neq S[i - M[i'] - 1]$. Além disso, $\textsc{Center}(S, i, M[i']) = \overline{\textsc{Center}(S, i', M[i'])}$, o que prova que $M[i] = M[i']$. Esse caso é o que ocorre na Figura 1.1, onde podemos determinar que, já que $M[i'] = 1 < g$, temos que $M[i] = 1$.

Se $M[i'] \geq g$, temos que $\textsc{Center}(S, i, g) = \overline{\textsc{Center}(S, i', g)}$ e este é um palíndromo, logo $M[i] \geq g$. Porém, não temos informação para delimitar superiormente o valor de $M[i]$ pois

a string $\textsc{Center}(S, i', M[i'] + 1)$ ultrapassa os limites de $\textsc{Center}(S, k, M[k])$, e portanto não pode ser comparada com $\textsc{Center}(S, i, M[i'] + 1)$. □

A proposição nos mostra uma forma de usar valores anteriormente calculados para otimizar o cálculo de $M[i]$. Note que no primeiro caso, quando $M[i'] < g$, conseguimos calcular $M[i]$ só usando valores calculados, e no segundo caso, quando $M[i'] \geq g$, podemos começar x de $k + M[k] - i$. Então, ao tentar maximizar o valor inicial de x, é interessante escolher k já calculado que maximize $k + M[k]$.

1.3 Algoritmo

Código 2 Cálculo dos alcances em tempo linear

```
1:  k = −1
2:  for i = 1 to |S| :
3:      if k ≠ −1 and k + M[k] > i and i + M[2k − i] < k + M[k] :        ▷ Caso (a)
4:          M[i] = M[2k − i]
5:      else
6:          if k ≠ −1 and k + M[k] > i :                                ▷ Caso (b)
7:              x = k + M[k] − i
8:          else                                                        ▷ Caso (c)
9:              x = 0
10:         while i − x > 1 and i + x < |S| and S[i − x − 1] = S[i + x + 1] :
11:             x += 1
12:         M[i] = x
13:         k = i
```

O algoritmo no Código 2 utiliza essas ideias para calcular o vetor M. Iremos mostrar que a variável k guarda um índice que maximiza o valor $k + M[k]$, ou -1 na primeira iteração.

1.3.1 Correção

O caso (a), quando a primeira parte da Proposição 1.2.1 vale, é identificado pelo **if** da linha 3. Nesse caso o valor de $M[i]$ já é calculado usando apenas $M[2k - i]$ e nada mais precisa ser feito.

Os dois casos restantes são: (b) quando a segunda parte Proposição 1.2.1 se aplica e (c) quando esta proposição não se aplica pois nenhum alcance anterior cobre i. Note que em ambos os casos temos uma cota inferior para x e então aumentamos x a partir desta cota. No caso (b), a cota é $k + M[k] - i$ e esse caso é tratado no **if** da linha 6. Note que se a condição desse **if** for satisfeita, então $i + M[2k - i] \geq k + M[k]$ (pois a condição do **if** da linha 3 não foi satisfeita), logo a segunda parte da proposição se aplica. No caso (c), a cota é 0 e esse caso é tratado no **else** da linha 8. Em ambos estes casos o **while** da linha 10 aumenta x enquanto possível, como no código anterior, e está correto pela Proposição 1.1.1.

Resta mostrar que o valor de k é atualizado corretamente. No caso (a) temos que

$$i + M[i] = i + M[2k - i] < i + k + M[k] - i = k + M[k],$$

logo o valor de k não deve ser atualizado. No caso (c), claramente k deve ser atualizado, e no caso (b) temos que

$$i + M[i] \geq i + k + M[k] - i = k + M[k],$$

pois pela segunda parte da Proposição 1.2.1 vale que $M[i] \geq k + M[k] - i$, logo podemos atualizar k. A linha 13 então atualiza k corretamente em ambos estes casos, e o código está correto.

1.3.2 Complexidade

Resta mostrar que o algoritmo leva tempo linear. O **for** da linha 2 realiza $|S|$ iterações, e, exceto pelo **while** da linha 10, todas as operações levam tempo constante, totalizando tempo linear.

Para analisar o tempo do **while**, vamos considerar como o valor de $k + M[k]$ muda durante o algoritmo. Quando tratamos o primeiro caso, este valor não muda. Ao tratar os outros casos, esse valor muda pois mudamos k para i na linha 13. Como discutido anteriormente, no caso (b) temos que $i + M[i] \geq i + k + M[k] - i = k + M[k]$, e no caso (c) temos que $i + M[i] \geq i > k + M[k]$ pois nesse caso o intervalo de k não cobre i. Logo, a atribuição na linha 13 nunca diminui o valor de $k + M[k]$.

Além disso, toda iteração do **while** aumenta o valor de x, e pela linha 12 sabemos que x ao final do **while** é o valor de $M[i]$. Então, devido à linha 13, cada iteração do **while** aumenta em 1 o valor de $k + M[k]$ ao final da iteração. Pela própria definição de alcance temos que $k + M[k] \leq |S|$, e portanto o número de iterações do **while** entre todas as iterações do **for** não excede $|S|$, e o tempo total consumido pelo algoritmo é $\mathcal{O}(|S|)$.

1.4 Palíndromos pares

Para conseguirmos informações sobre os palíndromos de tamanho par, é possível criar um vetor que na posição i guarda qual o maior palíndromo par centrado em i (um palíndromo par tem dois centros, então é preciso escolher um deles). As Proposições 1.1.1 e 1.2.1 podem ser adaptadas para palíndromos pares, e assim também o algoritmo.

Entretanto, é possível modificar a string S de forma que possamos usar o mesmo algoritmo (Código 2) para encontrar palíndromos pares. Para transformar um palíndromo par em um ímpar, é possível apenas adicionar um caractere qualquer após metade dos caracteres desse palíndromo. Esse novo caractere será o centro de um palíndromo ímpar. Como não sabemos o centro dos palíndromos, adicionamos um mesmo caractere entre todos os caracteres adjacentes.

Considere a string $\textsc{Fill}(S) = S[1]\$S[2]\$ \ldots \$S[|S|]$, ou seja, a string S com caracteres \$ inseridos entre todos os caracteres. Se $S[i\,..\,j]$ é um palíndromo, então a substring $\textsc{Fill}(S)[2i-1\,..\,2j-1]$ também é um palíndromo, e este palíndromo em $\textsc{Fill}(S)$ tem comprimento ímpar, independente se tinha comprimento par ou ímpar em S. Observe que, se o palíndromo era ímpar, então o centro do novo palíndromo continua o mesmo, mas se o palíndromo era par, o novo centro é o caractere \$ entre os dois centros originais.

Os novos palíndromos, entretanto, têm comprimento maior que na string original. Se M é o vetor de alcances calculado para a string $\textsc{Fill}(S)$, temos que, para todo $1 \leq i \leq |S|$, a posição i tem alcance $\left\lfloor \frac{M[2i-1]}{2} \right\rfloor$, ou seja, a substring $S[i - \left\lfloor \frac{M[2i-1]}{2} \right\rfloor \,..\, i + \left\lfloor \frac{M[2i-1]}{2} \right\rfloor]$ é o maior palíndromo

ímpar com centro em i. Além disso, para cada \$ em $\textsc{Fill}(S)$, ou seja, em toda posição par $i = 2k$ de $\textsc{Fill}(S)$, se $M[i] > 0$, então $S[k - \left\lceil \frac{M[i]}{2} \right\rceil + 1 \,..\, k + \left\lceil \frac{M[i]}{2} \right\rceil]$ é o maior palíndromo par de S com centros k e $k + 1$.

Com essas informações, é possível usar o algoritmo aplicado a $\textsc{Fill}(S)$ para ter a informação sobre todos os palíndromos de S. Note que $|\textsc{Fill}(S)| = 2|S| - 1$, então o consumo de tempo continua linear.

1.5 Usos

Com o vetor M calculado, é possível descobrir o tamanho da maior substring de S que é um palíndromo em tempo linear. Basta verificar, para todos os centros, qual o maior palíndromo com aquele centro. Além disso, dados dois índices i e j, é possível, em tempo constante, determinar se $S[i \,..\, j]$ é um palíndromo. Basta calcular qual seria o centro desse palíndromo, e verificar, na posição correspondente de M, se esta substring é um palíndromo.

Capítulo 2

Tries

2.1 Definição

Uma *arborescência* é uma árvore enraizada com arestas saindo da raiz, ou seja, um digrafo acíclico em que cada vértice tem grau de entrada no máximo 1, e existe um caminho da raiz para cada outro vértice. Dizemos que um vértice é uma *folha* se tem grau de saída 0. Dizemos que um vértice é *interno* se não é nem a raiz nem uma folha.

Definição. *Uma trie é uma arborescência na qual cada aresta está associada um caractere, e de um mesmo nó não saem duas arestas associadas ao mesmo caractere.*

A todo nó de uma trie associa-se uma string que consiste da junção dos caracteres de todas arestas no caminho da raiz a esse nó. As strings de todos os nós são diferentes.

Uma trie T é utilizada para armazenar um conjunto de strings $\mathcal{S}$. Para tanto, alguns nós de T são marcados e o conjunto de strings representado por T é o conjunto de strings associadas aos nós marcados.

Dizemos que o *tamanho* de um conjunto de strings $\mathcal{S} = \{S_1, \ldots, S_k\}$ é a soma dos tamanhos de suas strings, ou seja, $\sum_{i=1}^{k} |S_i|$.

2.2 Inserção

Se queremos que a string P tenha um nó associado a ela em T, existe um jeito único de adicioná-la à trie. Basta começar pela raiz de T e percorrer as arestas $P[1], \ldots, P[|P|]$, criando-as se necessário. Esse procedimento leva tempo $\mathcal{O}(|P||\Sigma|)$ ou $\mathcal{O}(|P| \lg |\Sigma|)$, dependendo da implementação.

Deve-se marcar os vértices associados a cada uma das strings de $\mathcal{S}$. Em geral estes são folhas, exceto se alguma string de $\mathcal{S}$ é prefixo de outra, como por exemplo no caso de `oi` na Figura 2.1.

2.3 Implementação

A grande dificuldade na implementação de uma trie T para $\mathcal{S}$ é decidir como guardar a arestas a partir de um vértice. O jeito mais simples é guardar, para cada nó, um vetor de $|\Sigma|$ posições com apontadores para outros nós, cada um associado a uma aresta. Os consumos de tempo mencionados

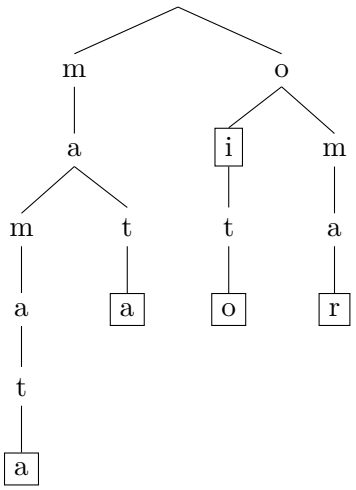


Figura 2.1: Uma trie T para $\mathcal{S}$ ={mamata, mata, oi, oito, omar}.

no futuro levarão em conta essa forma simples de representar uma trie. Essa implementação gasta espaço $\mathcal{O}(n|\Sigma|)$, onde n é o tamanho de $\mathcal{S}$. Nesse caso cada acesso a uma aresta (ou mesmo descobrir se uma aresta específica existe) leva tempo $\mathcal{O}(1)$.

Outra forma de guardar as arestas pode ser com uma lista ligada ou um vetor, guardando apenas a informação das arestas existentes, nesse caso a estrutura gastaria espaço $\mathcal{O}(n)$ mas os acessos levariam no pior caso tempo $\Theta(|\Sigma|)$. Uma forma melhor seria usar uma árvore de busca binária balanceada (ABBB), já que os caracteres são comparáveis. Isso consumiria espaço $\mathcal{O}(n)$ e tempo $\mathcal{O}(\lg|\Sigma|)$ por acesso.

Usamos que $v(c)$ é o filho do vértice v pelo caractere c, ou **null** se não existir tal filho. Usamos que **new** *node*() cria um novo nó com lista de adjacência vazia e não marcado, e que $u.mrk$ é um campo booleano que indica se tal nó é marcado. A função ADD(r, S) então adiciona a string P à trie com raiz r.

Código 3 Adição em trie

```
1: function ADD(r, P)
2:     v = r
3:     for c ∈ P[1..|P|] :
4:         if v(c) = null :
5:             v(c) = new node()
6:         v = v(c)
7:     v.mrk = true
```

Utilizando a implementação de listas de adjacência com vetores, é possível implementar **new** *node*() em tempo $\Theta(|\Sigma|)$, e então esse código consome tempo $\mathcal{O}(|P||\Sigma|)$. Começando r como um vértice vazio, e utilizando a função ADD(r, S) para toda string $S \in \mathcal{S}$, é então possível criar uma trie para $\mathcal{S}$ em tempo $\mathcal{O}(n|\Sigma|)$, onde n é o tamanho de $\mathcal{S}$.

A função no Código 3 funciona pois, como discutido na Seção 2.2, percorre o caminho indicado pela string P, caractere por caractere, e a linha 5 cria os vértices e arestas necessários quando este caminho não existe.

2.4 Utilidades

Com uma trie T para um conjunto de strings $\mathcal{S} = \{S_1, \ldots, S_k\}$, é possível, dada uma string P, saber se P é alguma das strings de $\mathcal{S}$ em tempo $\mathcal{O}(|P|)$, ou seja, independente do tamanho de $\mathcal{S}$. Para fazer isso, basta seguir em T o caminho $P[1], \ldots, P[|P|]$, se possível. Se for possível chegar ao final e o nó final for um dos nós marcados de T, então $P \in \mathcal{S}$. Se analisarmos o último nó que foi possível alcançar, este determina o maior prefixo comum de P e alguma string de $\mathcal{S}$.

A função $\text{LCP}(r, P)$ faz exatamente isso, determinando o maior prefixo comum da string P com a trie T de raiz r. Para descobrir se $P \in \mathcal{S}$, basta verificar se o retorno é $|P|$ e o último vértice alcançado está marcado.

Código 4 Maior prefixo comum em trie

```
1: function LCP(r, P)
2:     v = r
3:     for i = 1 to |P| :
4:         if v(P[i]) = null :
5:             return i − 1
6:         v = v(P[i])
7:     return |P|
```

Capítulo 3

Aho-Corasick

Neste capítulo vamos apresentar um algoritmo que recebe um conjunto de strings $\mathcal{S} = \{S_1, \ldots, S_k\}$ de tamanho n e uma string P, e determina, para cada $S_i \in \mathcal{S}$, quais são suas ocorrências em P. Seja T uma trie para o conjunto $\mathcal{S}$ e r sua raiz.

Com T, é possível descobrir quais strings de $\mathcal{S}$ são prefixos de P em tempo $\mathcal{O}(|P|)$. Com isso é possível encontrar todas as ocorrências de strings de $\mathcal{S}$ em P em tempo $\mathcal{O}(|P|^2)$, usando esse mesmo algoritmo para todos os sufixos de P.

O algoritmo apresentado nesse capítulo é conhecido como Aho-Corasick e foi criado por Alfred V. Aho and Margaret J. Corasick [AC75]. Com preprocessamento de T que consome tempo $\mathcal{O}(n|\Sigma|)$, similar ao preprocessamento do algoritmo KMP, conseguimos com esse algoritmo buscar todas as ocorrências das strings de $\mathcal{S}$ em P em tempo $\mathcal{O}(|P| + x)$, onde x é o número de ocorrências de strings de $\mathcal{S}$ em P.

3.1 Link de falha

Para cada nó v de T, defina $S(v)$ como a string associada a v em T, e $d(v) \coloneqq |S(v)|$ sua profundidade. Para cada nó $v \neq r$, seja $f(v)$ o nó u com maior $d(u)$ tal que $S(u) \sqsupset S(v)$. Ou seja, o nó de maior profundidade cuja string associada é sufixo próprio de $S(v)$.

Note que a string vazia associada à raiz é um sufixo próprio de todas outras strings, assim $f(v)$ existe para todo $v \neq r$. O valor $f(v)$ é chamado de *link de falha* de v.

A Figura 3.1 mostra a trie da Figura 2.1 com todos os links de falha que não apontam para a raiz desenhados. Note que um link de falha não necessariamente aponta para um ancestral do nó.

O seguinte algoritmo calcula corretamente o valor de $f(v)$, se v não é filho da raiz. Seja $c(v)$ o caractere associado à aresta que entra em v e $p(v)$ o pai de v em T. Também assuma que $v(c)$ é o filho de v por uma aresta de caractere c, ou **null** se esta não existir. Se v é filho da raiz, então $f(v) = r$.

Note que AuxiliaryLinks(v) precisa que os valores de f já estejam calculados para o pai de v, e todos os outros vértices com profundidade menor que a do pai de v. É possível conseguir isso se calcularmos os valores de f na mesma ordem que descobrirmos os vértices usando uma busca em largura (BFS) a partir da raiz. Quando estamos analisando um nó, todos com distância menor já foram analisados.

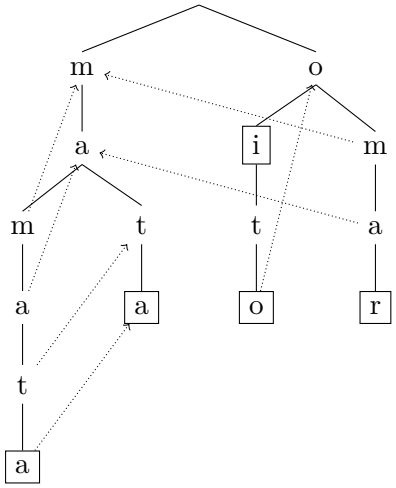


Figura 3.1: Trie T com links de falha.

```
1: function AuxiliaryLinks(v)
2:     f(v) = f(p(v))
3:     while f(v) ≠ r and f(v)(c(v)) = null :
4:         f(v) = f(f(v))
5:     if f(v)(c(v)) ≠ null :
6:         f(v) = f(v)(c(v))
```

3.2 Link de ocorrência

Para descobrir as ocorrências das strings de $\mathcal{S}$ em P, devemos processar um a um cada caractere de P. Após analisar o caractere $P[i]$, estaremos no nó v_i tal que $S(v_i)$ é o maior prefixo de alguma string de $\mathcal{S}$ que é sufixo de $P[1\,..\,i]$. Se para algum i temos que v_i está marcado em T, então temos uma ocorrência da string $S(v_i)$ que acaba na posição i de P. Mas essa não é a única situação em que temos ocorrências.

Se algum sufixo próprio de $S(v_i)$ for uma string em $\mathcal{S}$, então essa string também ocorre em P acabando na posição i, mesmo que o nó v_i não seja um nó marcado. Por exemplo, na trie da Figura 3.1, para o texto $P =$ `amamatax`, $S(v_7) =$ `mamata`, mas além da ocorrência dessa string, também existe a ocorrência de `mata` na mesma posição.

Para verificar esse tipo de ocorrência, para cada nó v de T, defina $N(v)$, o chamado *link de ocorrência*, como o nó u com $d(u)$ máximo tal que $S(u) \sqsupset S(v)$ e u está marcado, ou **null** se tal nó não existe. Ou seja, $N(v)$ é o primeiro vértice atingível "caminhando" pelos links de falha que é um nó marcado. Assim se, ao analisar $P[i]$, estamos no nó v_i e $N(v_i) \neq$ **null**, então uma ocorrência de $S(N(v_i))$ termina na posição i de P.

Dessa forma, "caminhando" pelos links de ocorrência de v_i, conseguimos determinar todas as strings de $\mathcal{S}$ que têm uma ocorrência em P terminando na posição i, em tempo proporcional ao número de tais ocorrências (pois cada link de ocorrência leva para uma ocorrência e leva tempo $\mathcal{O}(1)$ para ser percorrido).

Calcular esses links é fácil. Basta adicionar ao final da rotina AuxiliaryLinks(v)

```
7:    if f(v).mrk = true :
8:        N(v) = f(v)
9:    else
10:       N(v) = N(f(v))
```

3.3 Preprocessamento dos links auxiliares

A construção dos links de falha e de ocorrência pode ser feita da seguinte forma, dado que a trie para $\mathcal{S}$ já está construída.

Código 5 Preprocessamento do algoritmo de Aho-Corasick

```
1:  Q = {}                                                    ▷ fila vazia
2:  for c ∈ Σ :
3:      if r(c) ≠ null :
4:          Q.push(r(c))
5:          f(r(c)) = r
6:          N(r(c)) = null
7:  while Q ≠ {} :
8:      u = Q.pop()
9:      for c ∈ Σ :
10:         if u(c) ≠ null :
11:             AuxiliaryLinks(u(c))
12:             Q.push(u(c))
```

Teorema 3.3.1. *O algoritmo apresentado no Código 5 calcula corretamente f e N para todo nó v da trie T cuja raiz é r.*

Demonstração. Seja T a trie. O algoritmo realiza uma BFS nos nós de T, e aciona AuxiliaryLinks(v) do nó v no momento em que adiciona v à fila da BFS. Nesse momento, os links do pai de v e de todos vértices com distância à raiz menor já foram calculados.

Vamos usar que, se S é uma string, então $\textsc{Pop}(S) := S[1\,..\,|S|-1]$, ou seja, a string S removendo-se seu último caractere.

Vamos provar inicialmente que f é calculado corretamente. Como $\textsc{Pop}(S(v)) = S(u)$, onde u é o pai de v, temos que $\textsc{Pop}(S(f(v))) \supseteq S(f(u)) \supset S(u)$, ou seja, para encontrar $f(v)$, devemos encontrar o maior sufixo próprio de $S(u)$ cujo nó tem um filho com aresta associada a $c(v)$.

Queremos então mostrar que o **while** nas linhas 3-4 de AuxiliaryLinks(v) itera por todos os sufixos próprios de $S(u)$ que têm nós em T, em ordem decrescente de tamanho. Se mostrarmos isso, então o algoritmo escolhe o maior deles que também satisfaz a restrição adicional, e portanto a resposta está correta. Note que se a resposta for a raiz, como $d(v) > 1$, então não existe nenhum tal prefixo que satisfaça a segunda condição do **while**, logo o algoritmo também funciona corretamente.

Observe que $f(v)$ é inicializado com $f(u)$. Como assumimos que este está calculado corretamente, ele é o maior sufixo próprio de $S(u)$ em T pela definição de f. Note que $d(f(w)) < d(w)$, para todo nó w. Logo o **while** está iterando em ordem decrescente de tamanho dos sufixos. Como $f(w)$ é sempre o *maior* sufixo próprio de $S(w)$, nenhum sufixo de $S(u)$ vai deixar de ser visitado, logo o **while** itera por todos sufixos de $S(u)$ e calcula o valor de $f(v)$ corretamente.

Para calcular $N(v)$, devemos encontrar o primeiro nó entre $f(v), f(f(v)), \ldots, r$ que é marcado. Se $f(v)$ está marcado, então $N(v) = f(v)$, caso contrário, $N(f(v))$ é o primeiro nó marcado em $f(f(v)), \ldots, r$, e então é o primeiro nó marcado em $f(v), f(f(v)), \ldots, r$.

□

Complexidade. *O algoritmo apresentado no Código 5 calcula os valores de f e N para todos os nós em tempo $\mathcal{O}(n|\Sigma|)$.*

Demonstração. O laço da BFS claramente é executado uma vez para cada nó, e o laço interior é executado $|\Sigma|$ vezes. Logo o tempo é $\Omega(m|\Sigma|)$, onde m é o número de vértices da trie, que é $\mathcal{O}(n)$. Vamos mostrar que o cálculo dos valores de f e N levam no total tempo $\mathcal{O}(n)$. Assim o tempo total fica $\mathcal{O}(n|\Sigma|)$.

O cálculo de $N(v)$ claramente leva tempo $\mathcal{O}(1)$ para cada nó da trie. Para $S \in \mathcal{S}$, sejam $v_1, \ldots, v_{|S|}$ os nós associados a cada prefixo de S em T. Seja w_i^S o número de iterações do **while** nas linhas 3-4 de AuxiliaryLinks para calcular $f(v_i)$. Temos que $0 \leq d(f(v_i)) < d(v_i)$ para todo i, e $d(f(v_{i+1})) \leq d(f(v_i)) + 1$, pois o valor de f apenas "avança" no máximo uma vez por nó na linha 6 de AuxiliaryLinks(v). Além, para cada iteração do **while**, $d(f(v_i))$ diminui de pelo menos 1, e para isso é necessário que tenha aumentado por essa quantidade em iterações anteriores. Então temos que $\sum_{i=2}^{|S_i|} w_i^S \leq |S|$.

Dessa forma $\sum_{i=1}^{k} \sum_{j=2}^{|S|} w_j^{S_i} \leq \sum_{i=1}^{k} |S_i| = n$, e o tempo total para calcular f é $\mathcal{O}(n)$. Logo o tempo total do preprocessamento é $\mathcal{O}(n|\Sigma|)$. □

3.4 Busca

Como discutido no início da Seção 3.2, para determinar as ocorrências de $\mathcal{S}$ em P devemos, após processar o caractere $P[i]$, estar no nó v_i tal que $S(v_i) \sqsupseteq P[1\,..\,i]$ e $d(v_i)$ é máximo. Para isso usamos os links de falha, similarmente a como são usados no algoritmo AuxiliaryLinks(v). Ao processar $P[i]$, queremos encontrar o maior sufixo de $S(v_{i-1})$ que tem $P[i]$ como próximo caractere (em alguma string de $\mathcal{S}$). Para isso, basta usar os links de falha para iterar por todos esses sufixos.

Código 6 Busca do algoritmo Aho-Corasick

```
1:  u = r
2:  for c ∈ P[1..m] :
3:      while u ≠ r and u(c) = null :
4:          u = f(u)
5:      if u(c) ≠ null :
6:          u = u(c)
7:      if u.mrk = true :
8:          reportar ocorrência de u
9:      x = u
10:     while N(x) ≠ null :
11:         x = N(x)
12:         reportar ocorrência de x
```

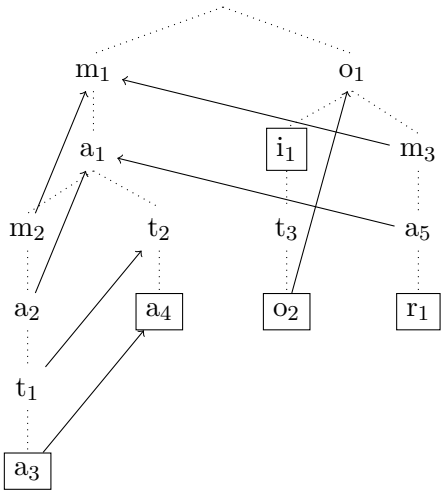


Figura 3.2: Links de falha.

3.4.1 Contagem de ocorrências

O algoritmo apresentado no Código 6 determina todas as ocorrências das strings de $\mathcal{S}$ em P, e leva tempo $\mathcal{O}(|P| + x)$, onde x é o número de tais ocorrências. Mas x é $\mathcal{O}(|P|k)$, onde k é o número de strings em $\mathcal{S}$. Um caso ruim por exemplo é $\mathcal{S} = \{\mathrm{a}, \mathrm{aa}, \ldots\}$ e $P = \mathrm{aaa}\ldots\mathrm{aa}$, mas é claro que existem muitos outros casos menos óbvios em que x é $\Theta(|P|k)$.

Por isso pode ser útil determinar apenas quais strings de $\mathcal{S}$ ocorrem no texto P, ou ainda mais informação, como o número de ocorrências de cada string de $\mathcal{S}$ em P. Adicionando um passo ao preprocessamento mostrado no Código 5 é possível desenvolver um algoritmo muito similar ao de busca que calcula o número de ocorrências de cada string de $\mathcal{S}$ em P em tempo $\mathcal{O}(n + |P|)$, onde n é o tamanho de $\mathcal{S}$. Note que esse tempo independe do número de ocorrências.

Para cada nó v de T que não é a raiz, crie uma nova variável $o(v)$, inicializada com 0, que ao final do algoritmo vai armazenar o número de ocorrências de $S(v)$ em P.

No Código 6, se substituirmos as linhas 7-12 pelo código abaixo, o cálculo dos valores $o(v)$ é feito corretamente, pois na i-ésima iteração v é o vértice com $d(v)$ máximo tal que $S(v) \sqsupseteq P[1\,..\,i]$, e dessa forma os vértices u de T tais que $S(u) \sqsupseteq P[1\,..\,i]$ são todos os vértices atingíveis por v usando os links de falha.

```
5:  ⋮
6:  x = u
7:  while x ≠ r :
8:      o(x) += 1
9:      x = f(x)
```

Observe que poderíamos, durante essa busca, apenas incrementar $o(u)$, e depois de feita a busca, "propagar" todos esses valores ao mesmo tempo pelos links de falha. A Figura 3.2 destaca os links de falha da trie da Figura 3.1, novamente sem mostrar os links para a raiz.

Considere um grafo T_f com os mesmos vértices da trie T, mas onde as arestas são os links de falha de T. Esse grafo é uma árvore, pois de cada nó exceto a raiz sai apenas um link de falha, e os

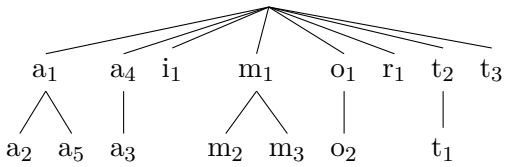


Figura 3.3: Árvore dos links de falha.

links de falha sempre apontam para vértices com valor de d menor, logo não há ciclos. A Figura 3.3 mostra como fica a árvore T_f do exemplo de trie da Figura 3.2.

Note que precisamos acumular os valores de cada nó para todos seus ancestrais nessa nova árvore, ou seja, para cada nó u, queremos calcular $o'(u) = \sum_v o(v)$ para todo v descendente de u em T_f. Um modo de fazer isso é, durante o preprocessamento de T, guardar uma lista de adjacências para cada cada vértice u que indica todos os vértices v tais que $f(v) = u$. Depois, com uma busca em profundidade, é possível acumular os valores de $o(u)$.

Porém, existe um jeito um pouco mais fácil de fazer isso, que dispensa guardar listas de adjacências para cada nó. Note que, no preprocessamento de T, visitamos seus nós em ordem da distância à raiz, e essa ordem é uma ordenação topológica da arborescência associada a T_f, pois os links de falha sempre apontam para nós com distância menor à raiz. Se durante o preprocessamento guardarmos essa ordem, podemos acumular os valores de $o(u)$ para todo nó u usando apenas um laço.

No Código 5, basta que, ao adicionar um nó na fila Q, também o adicionemos a um vetor *top*. Agora, *top* tem a ordenação topológica da arborescência associada a T_f. Então, para contar o número de ocorrências, basta usar esse vetor em um algoritmo similar ao do Código 6.

Código 7 Contagem de ocorrências

```
1:  u = r
2:  for c ∈ P[1..m] :
3:      while u ≠ r and u(c) = null :
4:          u = f(u)
5:      if u(c) ≠ null :
6:          u = u(c)
7:      o(u) += 1
8:
9:  for i = |top| down to 1 :
10:     o(f(top[i])) += o(top[i])
```

As linhas 1-7 contam, para cada nó u de T, quantas vezes o algoritmo de busca visita o nó u. As linhas 9-10 acumulam os valores de o. Podemos calcular o' usando a recorrência

$$o'(u) = o(u) + \sum_v o'(v)$$

onde a soma é sobre os v tais que $f(v) = u$.

Invariante. *No início da iteração i do* ***for*** *da linha 9, todos os valores de o' estão calculados corretamente levando em conta os vértices $top[i+1], \ldots, top[n-1]$, ou seja, para todo*

vértice u, $o'(u) = o(u) + \sum_v o'(v)$, *onde* $f(v) = u$ *e* $v \in \{top[i+1], \ldots, top[n-1]\}$.

Demonstração. Quando $i = n - 1$, o invariante vale pois as linhas 1-7 calculam corretamente os valores de o.

No começo da iteração i, se a invariante vale, o **for** apenas atualiza o valor de o' do vértice $f(top[i])$, mas esse é o único vértice que tem $top[i]$ como filho, logo o invariante continua valendo no início da próxima iteração. □

Então, no final do **for**, temos que os valores de o estão corretamente calculados. Note que não guardamos uma variável o', mas isso não influencia na corretude do código, pois esse valor fica guardado no próprio o, já que na primeira iteração $o'(u) = o(u)$, para todo u.

Capítulo 4

Árvores de sufixos

4.1 Introdução

4.1.1 Suffix tries

Tries são estruturas poderosas, mas guardam informação apenas sobre os prefixos das strings da coleção de strings que representam. Dada uma string P, se inserirmos todos os sufixos de P em uma trie, cada caminho nessa trie representará um *prefixo de um sufixo* de P, ou seja, uma substring de P. Se T é uma trie com todos os sufixos de P, dizemos que T é a *suffix trie* de P.

Dessa forma, se queremos saber se uma dada string S ocorre em P, basta checar se é possível percorrer o caminho $S[1], \ldots, S[|S|]$ na suffix trie de P. Note que isso leva tempo $\mathcal{O}(|S|)$, independente do tamanho de P.

4.1.2 Tries comprimidas

Definição. *Uma trie comprimida é uma arborescência na qual cada aresta tem um rótulo, que é uma sequência de caracteres, de um mesmo nó não saem duas arestas cujo rótulo começa com o mesmo caractere, e todo nó interno tem grau de saída pelo menos 2.*

Uma trie comprimida é uma generalização de tries em que as arestas podem ser rotuladas por strings em vez de só por caracteres. Para construir uma trie comprimida a partir de uma trie, "excluímos" os vértices internos com grau de saída 1, pois podemos juntar o rótulo da aresta que sai deste com o rótulo da aresta que entra nele. A Figura 4.1 mostra como ficaria a trie comprimida da Figura 2.1.

Note que a trie comprimida T' de T tem o mesmo número de folhas que T. Seja m esse número. Como todo vértice interno de T' tem pelo menos dois filhos, o número de vértices de T' é no máximo $2m$.

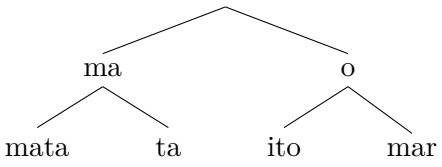


Figura 4.1: Exemplo de trie comprimida.

4.1.3 Suffix tries comprimidas

Uma suffix trie comprimida de P é chamada de *árvore de sufixos* de P. Note que, para tries comprimidas, apesar do número de vértices ser não mais que o dobro do número de strings na trie (pois cada string gera no máximo uma folha), o tamanho total das strings nas arestas ainda é proporcional ao tamanho total das strings na trie.

Porém, se temos uma árvore de sufixos de P, e uma aresta tem rótulo $P[i\,..\,j]$, então podemos guardar apenas os índices i e j nessa aresta e, assim, como o número de arestas é $\mathcal{O}(|P|)$, temos que a árvore de sufixos de P pode ser armazenada em espaço $\mathcal{O}(|P|.|\Sigma|)$, usando uma versão adaptada da representação discutida na Seção 2.3.

Queremos que cada sufixo de P esteja associado a uma folha dessa árvore, assim não teremos que marcar nós como em tries normais (haveria dificuldade já que nem todo nó da trie está representado nessa árvore). Um sufixo não tem uma folha associada quando é prefixo de outro sufixo. Uma forma simples de forçar isso é criar a árvore de sufixos da string $P\$$, onde \$ é um caractere que não ocorre na string P. Dessa forma nenhum sufixo é prefixo de outro sufixo e cada sufixo tem uma folha associada.

A Figura 4.2 mostra a árvore de sufixos da string `abracadabra`. Note que as arestas estão indicadas como se guardassem uma substring inteira, mas na prática vão guardar apenas dois índices. O rótulo da aresta uv é mostrado no vértice v.

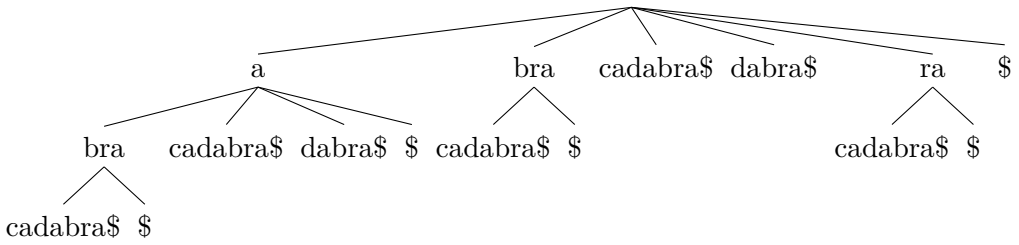


Figura 4.2: Árvore de sufixos para `abracadabra`.

4.2 Representação

Para construir a suffix trie de P, basta adicionar cada um dos sufixos de $P\$$ em uma trie. Mas se queremos criar diretamente a árvore de sufixos de P, existem dificuldades adicionais. É mais complicado navegar na árvore pois em certo ponto podemos estar "no meio" de uma aresta, e às vezes pode ser necessário "quebrar" uma aresta, para adicionar um vértice no meio desta.

Ao projetar algoritmos para tries, usávamos que $u(c)$ era o filho de u por uma aresta rotulada pelo caractere c, ou **null** se tal filho não existisse. Para tries comprimidas, usamos que $u.f[c]$ é o filho de u por uma aresta cujo rótulo tem *primeiro* caractere c, ou **null** se tal filho não existir. Além disso, é necessário guardar mais informação sobre as arestas, e para a aresta uv guardamos essa informação no vértice v, ou seja, em cada nó guardamos os dados sobre a aresta incidente neste.

Seja $e = uv$ a aresta que chega em v. Guardamos em $v.p$ o identificador de u. Os campos $v.l$ e $v.r$ guardam os índices que determinam o rótulo $P[l\,..\,r]$ da aresta e. Baseados nesses valores, escrevemos $v.s[i] = P[v.l + i - 1]$ para acessar o i-ésimo caractere do rótulo de e, e $len(v.s) = v.r - v.l + 1$

para o tamanho do rótulo de e, para deixar o código mais limpo. Usamos **new** $node(l, r, p)$ para representar a criação de uma aresta que sai de p e tem rótulo $P[l\,..\,r]$, se ligando a um novo vértice. Como convenção a raiz é inicializada com **new** $node(1, 0,$ **null**$)$, ou seja, sua "aresta" está rotulada pela string vazia.

Seja T uma trie, e T_c a trie comprimida associada. Para navegar em T, é suficiente guardar uma variável u que representa o nó atual que o algoritmo está analisando. Em T_c usaremos duas variáveis, cn e cd, onde cn é um vértice em T_c, e cd um inteiro que determina quantos caracteres do rótulo da aresta de $cn.p$ para cn já percorremos. O par (cn, cd) determina unicamente um vértice em T. Note que se $cd = len(cn.s)$ então estamos exatamente no vértice cn, e não no meio de uma aresta.

4.3 Construção em tempo quadrático

O algoritmo no Código 8 mostra como construir a árvore de sufixos de P em tempo $\mathcal{O}(|P|^2)$, usando memória $\mathcal{O}(|P|.|\Sigma|)$. Apresentamos esse código aqui pois pode ser útil para strings pequenas, já que é mais fácil de entender e implementar que o código que vamos apresentar na Seção 4.5, que consome tempo linear. Ele também deixa claro como utilizamos a forma de representação discutida na Seção 4.2.

A função BuildSuffixTreeQuad(P) constrói a árvore de sufixos da string P, adicionando cada sufixo de P em uma trie comprimida. A árvore de sufixos inicialmente consiste só da raiz r, inicializada como discutido acima.

Código 8 Árvore de sufixos em tempo quadrático

```
1:  function BuildSuffixTreeQuad(P)
2:      P += '$'
3:      for i = 1 to |P| :
4:          cn = r
5:          cd = 0
6:          for j = i to |P| :
7:              if cd = len(cn.s) and cn.f[P[j]] = null :        ▷ Não é necessário quebrar a aresta
8:                  cn.f[P[j]] = new node(j, |P|, cn)
9:                  break
10:             if cd < len(cn.s) and cn.s[cd + 1] ≠ P[j] :      ▷ Quebrando a aresta
11:                 mid = new node(cn.l, cn.l + cd − 1, cn.p)
12:                 cn.p.f[mid.s[1]] = mid
13:                 mid.f[cn.s[cd + 1]] = cn
14:                 cn.p = mid
15:                 cn.l += cd
16:                 mid.f[P[j]] = new node(j, |P|, mid)
17:                 break
18:             if cd = len(cn.s) :                              ▷ Percorrendo 1 caractere
19:                 cn = cn.f[P[j]]
20:                 cd = 0
21:             cd += 1
```

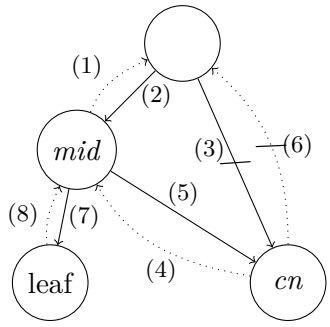


Figura 4.3: Inserção de *mid*.

Invariante. *No início da iteração j do* ***for*** *da linha 6, o par (cn, cd) indica o nó (na trie não comprimida) que tem como string associada $P[i\,..\,j-1]$.*

Demonstração. Quando $j = i$, o invariante vale pois a raiz é associada à string vazia e o par (cn, cd) é iniciado corretamente nas linhas 4-5.

Suponha que o invariante vale no início da iteração j. Os **if**s das linhas 7 e 10 verificam se não existe caminho a partir do nó atual com caractere $P[j]$, e se algum deles executar, o **for** é terminado.

Se os **if**s não executarem, então se $cd < len(cn.s)$ sabemos que o $(cd+1)$-ésimo caractere na aresta incidente em cn é $P[j]$, e assim podemos incrementar cd em 1 para representar o vértice associado a $P[i\,..\,j]$. Se $cd = len(cn.s)$ sabemos que existe aresta com primeiro caractere $P[j]$ saindo de cn, assim as linhas 19-20 fazem cn apontar para o vértice no final dessa aresta, e cd é inicializado com 0 e depois incrementado para 1, assim representando a string $P[i\,..\,j]$.

Assim os novos valores de (cn, cd) indicam o nó associado a $P[i\,..\,j]$, e o invariante vale na próxima iteração. □

Pelo invariante, temos que o **for** nas linhas 6-21 percorre o maior prefixo de $P[i\,..\,|P|]$ que já está na árvore de sufixos. Se este terminar normalmente, então o sufixo $P[i\,..\,|P|]$ já está na árvore de sufixos, senão os casos são tratados pelos **if**s das linhas 7 e 10 da seguinte forma:

Linha 7. Se $cd = len(cn.s)$ e nenhuma aresta sai de cn com primeiro caractere $P[j]$ então não existe caminho, e nesse caso basta criar uma nova folha apontada por cn associada ao sufixo $P[j\,..\,|P|]$.

Linha 10. Se $cd < len(cn.s)$ e o próximo caractere da aresta incidente em cn é diferente de $P[j]$ então não existe caminho, e nesse caso é necessário quebrar esta aresta em duas neste ponto, sendo uma para os primeiros cd caracteres e outra para o restante. A partir do novo vértice, cria-se uma aresta associada ao sufixo $P[j\,..\,|P|]$. Essa quebra de aresta é a parte mais complicada do algoritmo, e será explicada usando a Figura 4.3. Os links guardados na lista de adjacência (variável f) de cada vértice são representadas com arestas cheias, e os links de pai guardadas na variável p são representadas por arestas pontilhadas.

Inicialmente apenas os links (3) e (6) existem. A linha 11 cria o vértice *mid* e o link (1). A linha 12 apaga o link (3) e cria o link (2), pois apesar de criarmos *mid* como filho de $cn.p$, na lista de adjacência desse vértice não é atualizada pela função **new** *node*. As linhas 13 e 14 fazem cn ser filho de *mid*, criando o link (5), apagando o link (6) e criando o link (4).

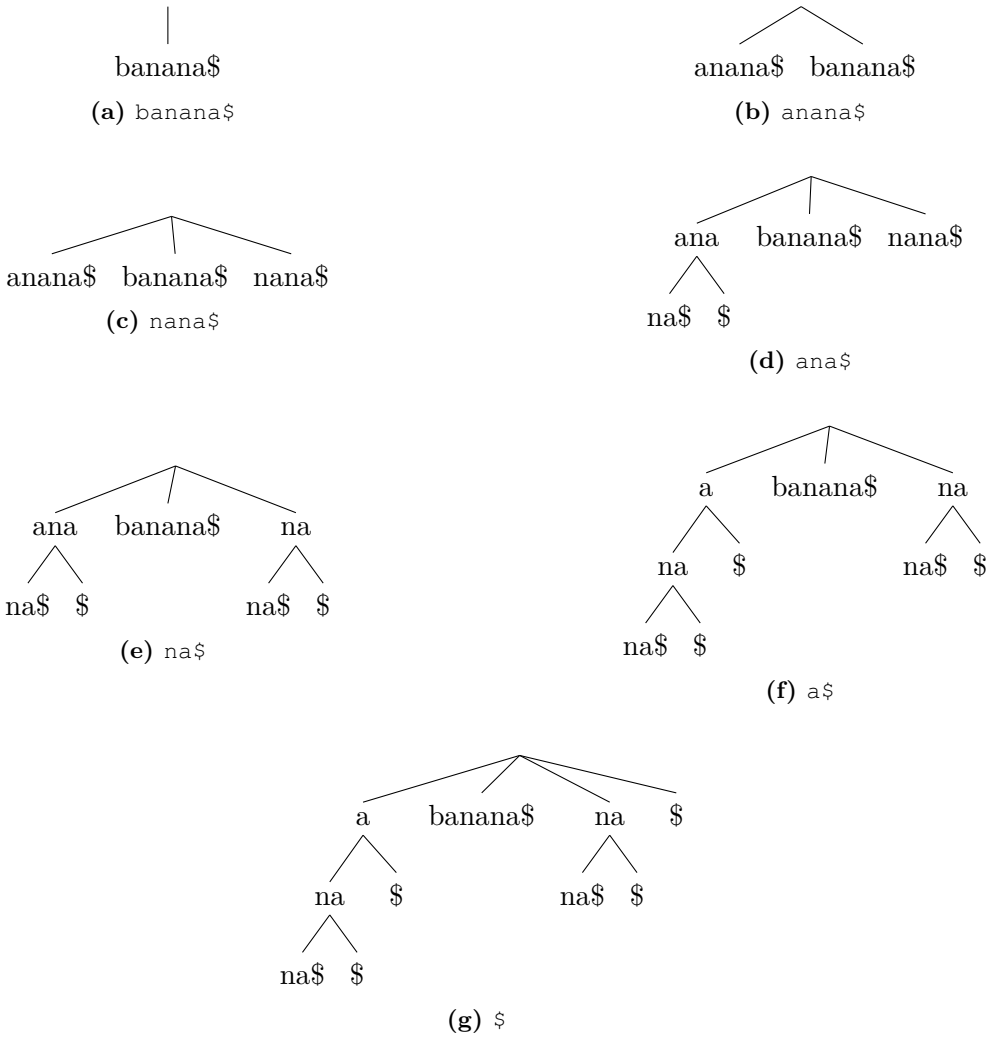


Figura 4.4: Construção da árvore de sufixos de `banana` em tempo quadrático.

A linha 15 apenas atualiza o rótulo da aresta que chega em cn, removendo os cd primeiros caracteres, e finalmente a linha 16 cria a nova folha e assim os links (7) e (8).

Logo, o sufixo $P[i\,..\,|P|]$ é adicionado corretamente à árvore de sufixos. Como todos os sufixos são adicionados pelo **for** da linha 3, então a função BuildSuffixTreeQuad(P) funciona corretamente. Note que como adicionamos um caractere \$ ao final de P, nenhum sufixo é prefixo de outro sufixo, logo uma destas condições sempre acontece ao adicionar cada sufixo. A Figura 4.4 mostra a construção de uma árvore de sufixos para `banana` usando o algoritmo apresentado nessa seção, adicionando sufixo por sufixo.

4.4 Uso

O Código 8 também mostra como percorrer a árvore de sufixos, e com isto já é possível determinar se S ocorre em P em tempo $\mathcal{O}(|S|)$. Basta verificar se é possível seguir o caminho $S[1], \ldots, S[|S|]$ na árvore de sufixos de P. Se em cada vértice da árvore de sufixos guardarmos o número de folhas

que estão em sua subárvore (pode ser facilmente feito com uma busca em profundidade (DFS) em tempo $\mathcal{O}(|P|.|\Sigma|)$), também teremos o número de ocorrências de S em P, pois cada folha alcançável indica uma ocorrência no começo do sufixo associado a esta folha. Aqui, estamos usando o fato de nossa árvore ter uma folha para cada sufixo de P. Note que se o final de S está no meio de uma aresta, o número de folhas atingíveis é o mesmo que se seguirmos essa aresta até o final.

É possível ainda descobrir quais são essas ocorrências em tempo proporcional ao número destas, apenas percorrendo (com uma DFS, por exemplo) a subárvore do vértice que representa S, pois o número de nós nesta subárvore é proporcional ao número de folhas. Observe que devemos percorrê-la como uma árvore normal, ignorando que as arestas podem ter um rótulo extenso, assim cada aresta é percorrida em tempo $\mathcal{O}(1)$ e o tempo de visitar as folhas fica proporcional ao número destas.

Note que se tivermos uma árvore de sufixos de P, conseguimos realizar o mesmo que com o algoritmo de Aho-Corasick, pois para um conjunto $\mathcal{S} = \{S_1, \ldots, S_k\}$ de strings, podemos descobrir as ocorrências de cada string de $\mathcal{S}$ em tempo $\mathcal{O}(\sum_{i=1}^{k} |S_i| + x)$, onde x é o número dessas ocorrências, usando o algoritmo descrito acima para cada uma das strings. Na próxima seção apresentaremos a criação de uma árvore de sufixos em tempo linear, e assim será possível atingir o mesmo tempo que Aho-Corasick, sem precisar ter acesso às strings de $\mathcal{S}$ de antemão. O código, contudo, é mais complicado que o do Aho-Corasick.

4.5 Construção em tempo linear

O primeiro algoritmo para criação de árvores de sufixos em tempo linear foi proposto por Weiner [Wei73], mas o algoritmo que apresentaremos será o algoritmo de Ukkonen [Ukk95], por ser mais simples.

A ideia deste algoritmo é adicionar um prefixo de P por vez, segundo a ordem $P[1\,..\,1], P[1\,..\,2], \ldots, P[1\,..\,|P|]$. Dessa forma temos um algoritmo *online*, ou seja, não é necessário ter a string P inteira inicialmente. Ela pode ser dada caractere por caractere.

Vamos definir T_j como a árvore de sufixos para $P[1\,..\,j]$. Na j-ésima iteração, temos T_{j-1} e queremos, a partir desta, construir T_j. Para isso adicionamos os sufixos $P[1\,..\,j], P[2\,..\,j], \ldots, P[j\,..\,j]$, nesta ordem. Para $1 \leq i \leq j$, ao adicionarmos $P[i\,..\,j]$, dizemos que estamos realizando a i-ésima extensão na j-ésima iteração. Note que, pela descrição, esse algoritmo leva tempo $\mathcal{O}(n^3)$, potencialmente pior que o algoritmo simples descrito na seção anterior. Mas com algumas otimizações é possível diminuir seu tempo para $\mathcal{O}(n)$.

Definição. *Dizemos que uma string S **está** na árvore de sufixos T se existe um caminho a partir da raiz de T no qual a string associada é S.*

Lema 4.5.1. *Se alguma string S está na árvore de sufixos T, então todos os sufixos de S também estão.*

Demonstração. Como T é uma árvore de sufixos de alguma string P, a string S é uma substring de P (pois todo caminho a partir da raiz de T representa uma substring de P). Mas todo sufixo de S também é uma substring de P, e assim está representado em T. □

Lema 4.5.2. *Para $1 \leq i \leq j$, se o nó associado a $P[i\,..\,j]$ é uma folha em T_j, então $P[i\,..\,j+1]$ não está representado em T_j. Ao adicionar $P[i\,..\,j+1]$, esse também será associado a uma folha no final da $(j+1)$-ésima iteração.*

Demonstração. Como $P[i\,..\,j] \sqsubset P[i\,..\,j+1]$, o caminho para $P[i\,..\,j+1]$ passa pelo nó associado a $P[i\,..\,j]$. Mas este é uma folha, logo $P[i\,..\,j+1]$ não está em T_j.

Ao ser adicionado, $P[i\,..\,j+1]$ é uma folha, pois basta adicionar o caractere $P[j+1]$ na aresta que chega na folha de $P[i\,..\,j]$. Para mostrar que este vértice continua uma folha ao final da $(j+1)$-ésima iteração, basta mostrar que nenhuma outra string adicionada nessa iteração tem $P[i\,..\,j+1]$ como prefixo. Suponha que $P[i\,..\,j+1] \sqsubset P[k\,..\,j+1]$, para algum k. Então $P[i\,..\,j] \sqsubset P[k\,..\,j]$. Mas $P[i\,..\,j]$ é uma folha, e $P[k\,..\,j]$ é uma string maior que $P[i\,..\,j]$ que tem esta como prefixo, o que leva a uma contradição pois $P[k\,..\,j]$ já estava em T_j. □

Ao executar a i-ésima extensão na j-ésima iteração, estamos adicionando a substring $P[i\,..\,j]$ a T_{j-1}. Então claramente já existe caminho associado a $P[i\,..\,j-1]$. Logo resta apenas adicionar o caractere $P[j]$ a esse caminho. Existem quatro casos possíveis:

Caso 1. O caminho $P[i\,..\,j]$ já existe em T_{j-1}. Nesse caso nada precisa ser feito.

Caso 2. O caminho $P[i\,..\,j-1]$ termina em um vértice não-folha. Nesse caso basta criar uma folha conectada a esse vértice por uma aresta associada a $P[j]$.

Caso 3. O caminho $P[i\,..\,j-1]$ termina em uma aresta (pois as arestas de uma árvore de sufixos podem ser associadas a rótulos com mais de um caractere). Nesse caso, é necessário "quebrar" a aresta, ou seja, criar um vértice interno no meio dessa aresta e, a partir deste ponto, é como no caso anterior.

Caso 4. O caminho $P[i\,..\,j-1]$ termina em uma folha. Nesse caso basta apenas "estender" essa folha, ou seja, aumentar em um o tamanho da aresta que chega nessa folha.

Teorema 4.5.3. *Para $1 \leq i \leq j$, seja $C(i,j)$ o caso que ocorreu na i-ésima extensão da j-ésima iteração. Para todo $1 \leq j \leq |P|$, a sequência $C(1,j), C(2,j), \ldots, C(j,j)$ se comporta da seguinte maneira:*

- *Há um prefixo de 4's.*
- *Após esse prefixo, há uma sequência de 3's ou 2's.*
- *Após essa sequência, há um sufixo de 1's.*

Além disso, para $j > 1$, $C(i,j) = 4$ se e somente se $C(i,j-1) \neq 1$.

Demonstração. Vamos provar o teorema por indução em j, ou seja, vamos provar por indução em j que a sequência $C(1,j), \ldots, C(j,j)$ satisfaz os três itens do teorema e a observação que segue. A base, $j = 1$, é fácil: a árvore T_0 só tem a raiz então $C(1,1) = 2$, pois considera-se que a raiz não é uma folha, mesmo quando sozinha.

Para $j > 1$, suponha por indução que $C(1,j-1), \ldots, C(j-1,j-1)$ satisfaz as condições do teorema. Vamos mostrar que as condições valem para j. Para $1 \leq i < j$, se $C(i,j-1) = 4$, pelo

Lema 4.5.2 temos que $C(i,j) = 4$ também. O mesmo vale se $C(i,j-1) \in \{2,3\}$, pois esses casos levam à criação de folhas. Então o prefixo de 4's, 3's e 2's de $C(1,j-1),\ldots,C(j-1,j-1)$ correspondem a um prefixo de 4's de $C(1,j),\ldots,C(j-1,j)$.

Para $1 \leq i \leq j$, o Lema 4.5.1 nos diz que se $P[i\,..\,j]$ está na árvore de sufixos, então $P[i+1\,..\,j],\ldots,P[j\,..\,j]$ também estão. Isso é equivalente a dizer que se $C(i,j) = 1$, então $C(i+1,j) = \cdots = C(j,j) = 1$, logo, a sequência de 1's forma um sufixo.

Note que resta apenas mostrar que $C(i,j) \neq 4$ se $C(i,j-1) = 1$ ou se $i = j$. Se $i = j$, isso vale pois $P[i\,..\,i-1]$ termina na raiz, que não é uma folha. Se $i < j$ então, se $C(i,j-1) = 1$, temos que a string $P[i\,..\,j-1]$ já estava na árvore T_{j-2} quando foi inserida na i-ésima extensão da $(j-1)$-ésima iteração. Assim $P[i\,..\,j-1] = P[i-k\,..\,j-1-k]$, para algum $k > 0$, e nesse caso $P[i-k\,..\,j-1-k] \sqsubset P[i-k\,..\,j-1]$. Logo $P[i\,..\,j-1]$ não é uma folha e $C(i,j) \neq 4$. □

Com esse teorema, é possível realizar apenas $\mathcal{O}(|P|)$ extensões para criar $T_{|P|}$. As operações feitas no caso 4 podem ser feitas automaticamente. Basta, nas folhas, em vez de guardar l e r normalmente para indicar a string $P[l\,..\,r]$, no valor de r guardarmos um valor especial (como -1 ou algum valor maior que $|P|$) indicando que o valor real de r é o número da iteração atual. Porém, só é necessário que o comprimento do rótulo sempre seja maior ou igual ao valor que teria se r fosse igual ao número da iteração atual. Logo, podemos guardar $|P|$ nos valores de r da folha. O algoritmo funcionará e esses campos estarão corretos quando o algoritmo acabar.

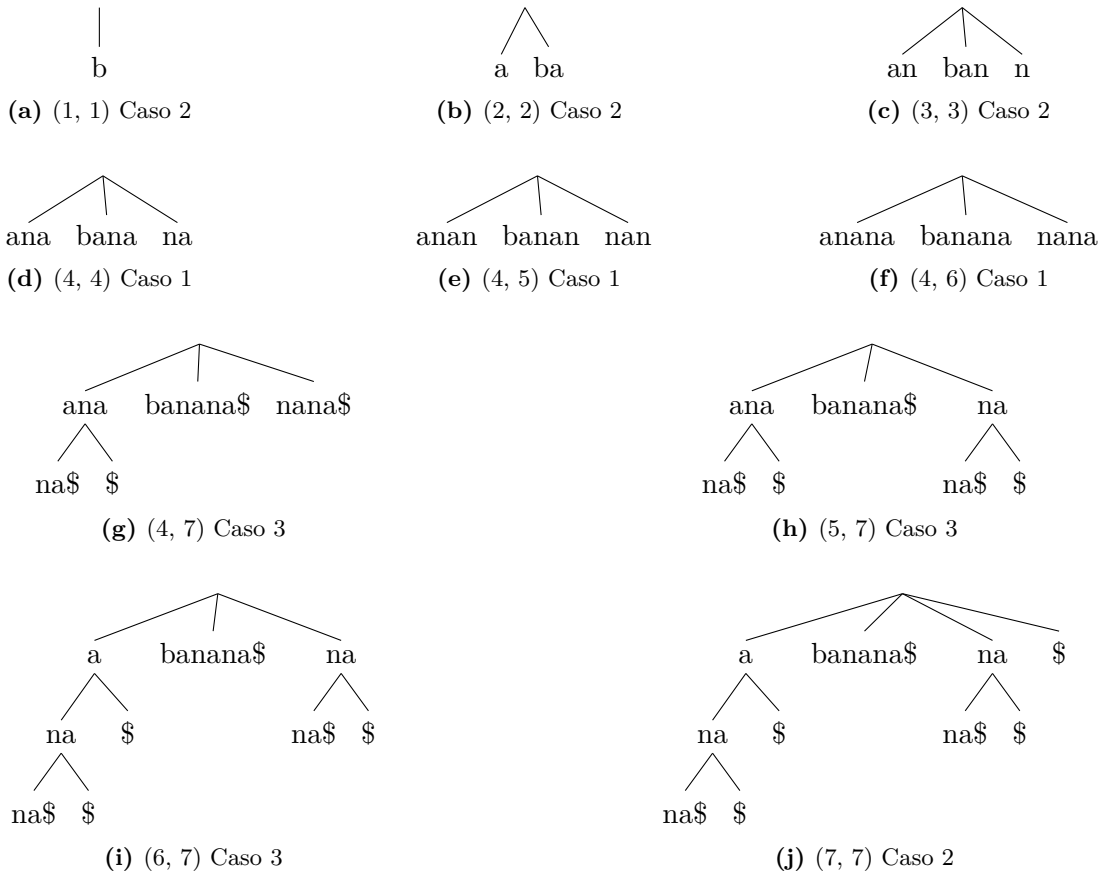
Dessa forma, não é necessário fazer nada no caso 4. Precisamos apenas começar da primeira ocorrência do caso 1 da iteração $j-1$, ou de j se o caso 1 não tiver ocorrido na iteração $j-1$. Quando ocorre um caso 1 na iteração j, sabemos que todas as próximas extensões nessa iteração serão do caso 1. Portanto não é necessário executar o resto da iteração e podemos ir para a iteração $j+1$. Assim, no máximo um índice de extensão é analisado em duas iterações consecutivas (se ocorreu caso 1 na primeira destas iterações), e executamos no máximo $2|P|$ extensões entre todas as iterações.

O Código 9 mostra como criar uma árvore de sufixos para uma string P em tempo linear. O **for** na linha 5 executa cada uma das iterações e o **while** na linha 6 executa cada uma das extensões. A otimização discutida acima está implementada, e assim são feitas apenas $\mathcal{O}(|P|)$ extensões. As variáveis *cn* e *cd* são usadas como anteriormente para navegar pela árvore. O novo campo $v.suf$, chamado de *link de sufixo* de v, representa, para um vértice interno (não-folha e não-raiz) v associado a $P[i\,..\,j]$, o vértice u associado a $P[i+1\,..\,j]$, ou seja, à string de v sem o primeiro caractere. Esse valor é usado para navegar mais rapidamente pela árvore, como mostraremos adiante. A variável *ns* é uma variável auxiliar na criação dos links de sufixo, e indica o único nó interno que está com o valor de *suf* indefinido, quando há um.

A Figura 4.5 mostra a construção da árvore de sufixos de `banana` com o algoritmo do Código 9, mostrando o resultado depois de cada extensão realizada. Os números em parênteses mostram respectivamente a extensão e a iteração atual. A árvore é mostrada com as aplicações do caso 4 implícitas, ou seja, apesar de as folhas terem valor de r como $|P|$, mostramos como se o valor fosse o da iteração atual.

Código 9 Árvore de sufixos em tempo linear

```
1: function BuildSuffixTree(P)
2:     P += '$'
3:     r = new node(1, 0, null)                          ▷ raiz inicializada com rótulo vazio
4:     (i, cn, cd, ns) = (1, r, 0, null)
5:     for j = 1 to |P| :                                ▷ j-ésima iteração
6:         while i ≤ j :                                 ▷ i-ésima extensão
7:             if cd = len(cn.s) and cn.f[P[j]] ≠ null :  ▷ caso 1
8:                 cn = cn.f[P[j]]
9:                 cd = 0
10:            if cd < len(cn.s) and cn.s[cd + 1] = P[j] :
11:                cd += 1
12:                break
13:            if cd = len(cn.s) :                       ▷ caso 2
14:                cn.f[P[j]] = new node(j, |P|, cn)
15:                if cn ≠ r :
16:                    cn = cn.suf
17:                    cd = len(cn.s)
18:            else                                      ▷ caso 3
19:                mid = new node(cn.l, cn.l + cd − 1, cn.p)
20:                cn.p.f[mid.s[1]] = mid
21:                mid.f[cn.s[cd + 1]] = cn
22:                cn.p = mid
23:                cn.l += cd
24:                mid.f[P[j]] = new node(j, |P|, mid)
25:                if ns ≠ null :
26:                    ns.suf = mid
27:                cn = mid.p
28:                if cn ≠ r :
29:                    cn = cn.suf
30:                    g = j − cd
31:                else                  ▷ No início do laço, cn está associado a P[i + 1 .. g − 1]
32:                    g = i + 1
33:                while g < j and g + len(cn.f[P[g]].s) ≤ j :
34:                    cn = cn.f[P[g]]
35:                    g += len(cn.s)
36:                if g = j :
37:                    ns = null
38:                    mid.suf = cn
39:                    cd = len(cn.s)
40:                else
41:                    ns = mid
42:                    cn = cn.f[P[g]]
43:                    cd = j − g
44:            i += 1
45:    return r
```



(a) (1, 1) Caso 2 **(b)** (2, 2) Caso 2 **(c)** (3, 3) Caso 2

(d) (4, 4) Caso 1 **(e)** (4, 5) Caso 1 **(f)** (4, 6) Caso 1

(g) (4, 7) Caso 3 **(h)** (5, 7) Caso 3

(i) (6, 7) Caso 3 **(j)** (7, 7) Caso 2

Figura 4.5: Construção da árvore de sufixos de `banana` em tempo linear.

Invariante. *No começo da i-ésima iteração do* ***while*** *da linha 6, na j-ésima iteração do* ***for****, o par (cn, cd) indica o vértice (ou aresta) que representa a substring $P[i\,..\,j-1]$. Nesse ponto, a árvore de sufixos é T_{j-1} com as strings $P[1\,..\,j], \ldots, P[i-1\,..\,j]$ adicionadas.*

Além disso, sempre que um vértice interno for criado (por uma aplicação do caso 3), o seu link de sufixo será definido antes do final da próxima extensão. Enquanto um nó tiver o valor de suf indefinido, seu valor será guardado na variável ns.

Demonstração. Na primeira iteração do **while** da linha 6, estamos na primeira extensão da primeira iteração, ou seja, queremos adicionar a substring $P[1\,..\,1]$, e o par (cn, cd) inicialmente indica a raiz, que está associada à string vazia, e o invariante vale.

No começo da iteração, estamos na substring $P[i\,..\,j-1]$ e queremos adicionar o caractere $P[j]$. Como discutido anteriormente, isso se reduz a quatro casos. As linhas 7-12 tratam o caso 1; as linhas 13-17 tratam o caso 2; as linhas 18-43 tratam o caso 3; o caso 4 nunca é tratado explicitamente, pois sempre estamos em uma extensão em que ocorre um dos demais casos.

Se $cd < len(cn.s)$, dizemos que a substring representada pelo par (cn, cd) é interna, ou seja, acaba "no meio" de uma aresta.

O caso 1 ocorre quando a substring $P[i\,..\,j-1]$ é interna e a aresta à qual pertence continua com o caractere $P[j]$, tratado nas linhas 10-12, ou quando $P[i\,..\,j-1]$ não é interna e o vértice em que acaba tem uma aresta que começa com $P[j]$. Nesse caso seguimos "0" caracteres na direção dessa aresta (linhas 7-9) e o caso pode ser tratado como se fosse o anterior. Se ocorrer o caso 1, a substring $P[i\,..\,j]$ já existe na árvore, e a iteração j acaba. A próxima extensão vai ser a i-ésima extensão da $(j+1)$-ésima iteração (pela otimização discutida anteriormente), ou seja, vamos adicionar a string $P[i\,..\,j+1]$. Para manter o invariante, é necessário avançar para a string $P[i\,..\,j]$, ou seja, basta seguir o caminho que sabemos que existe, ou continuando na aresta se $P[i\,..\,j-1]$ é interno, ou seguindo pela aresta que começa com $P[j]$, que é feito pelo incremento de cd na linha 11.

Para o caso 2, as linhas 13-17 apenas criam uma nova folha. Para manter o invariante na próxima extensão (que adicionará a substring $P[i+1\,..\,j]$), movemos de $P[i\,..\,j-1]$ para $P[i+1\,..\,j-1]$ usando $cn.suf$, se cn for interno. Se cn for a raiz basta continuar nesta. Se cn for interno, o seu campo suf já está definido pois cn não foi criado nesta extensão.

O caso 3 é mais complexo. Nele, $P[i\,..\,j-1]$ é interno e não continua com o caractere $P[j]$, ou seja, o $(cd+1)$-ésimo caractere da aresta incidente em cn é diferente de $P[j]$, logo é necessário separar a aresta entre seus primeiros cd caracteres e o restante, e deste vértice novo criar uma nova folha. Esse procedimento ocorre nas linhas 19-24 e é o mesmo que utilizado nas linhas 11-16 do Código 8, e já foi explicado em sua explicação.

As linhas 25-26 atualizam o link de sufixo de ns, se necessário, e estão corretas pois assumimos que um vértice fica sem link de sufixo no máximo uma extensão, então se $ns \neq$ **null**, este é da extensão $i-1$, associado a $P[i-1\,..\,j-1]$, e seu link de sufixo deve apontar para mid, que é associado a $P[i\,..\,j-1]$. Perceba que a última extensão de uma iteração nunca deixa $ns \neq$ **null**, pois ou é do caso 1, ou adicionou a string $P[k\,..\,k]$, onde k é o número dessa iteração, logo não cria vértice interno.

Assim a string $P[i\,..\,j]$ é adicionada à árvore de sufixos, e agora é necessário navegar para a string $P[i+1\,..\,j-1]$, para manter o invariante para a string $P[i+1\,..\,j]$ a ser adicionada na próxima extensão, mas mid ainda não tem um link de sufixo, então não é tão simples quanto no

caso 2. Se o pai de *mid* for um vértice interno, podemos começar a busca por $P[i\,..\,j]$ a partir do link de sufixo do pai de *mid*, já que é um prefixo de $P[i+1\,..\,j-1]$ (pois é um prefixo de $P[i\,..\,j-1]$ retirando o primeiro caractere). As linhas 27-43 fazem essa busca. Nas linhas 27-32, inicializamos *cn* com o vértice inicial da busca, tratando o caso de o pai de *mid* ser a raiz (e não ter link de sufixo). Note que nessa busca não utilizaremos a variável *cd*. Sempre assumimos que estamos no vértice *cn*, independente do valor de *cd*.

O invariante do **while** da linha 33 é que g indica que *cn* está em um vértice associado a $P[i+1\,..\,g-1]$, assim queremos aumentar g até que $g = j$. A base vale pois as linhas 30 e 32 iniciam o valor de g corretamente. A linha 30 trata o caso em que a busca começa na raiz, então inicialmente a string a raiz representa $P[i+1\,..\,i]$. A linha 32 trata o caso em que a busca começa em $v.p.suf$, onde v é o vértice em que iniciamos a extensão (o valor de *cn* no começo da extensão). Nesse caso, como v está associado a $P[i\,..\,j-1]$, temos que $v.p$ está associado a $P[i\,..\,j-1-cd]$, e $v.p.suf$ a $P[i+1\,..\,j-1-cd]$. Então g é corretamente inicializado com $j - cd$.

Dado que o invariante vale no começo da iteração, se a condição do **while** for verdadeira, ou seja, ainda não alcançamos $g = j$, e percorrer inteiramente a aresta saindo de *cn* com primeiro caractere $P[g]$ não faz g exceder j, então *cn* é mudado para $cn.f[P[g]]$, ou seja, para o filho por uma aresta que começa com $P[g]$. Desse modo, g é incrementado na linha 35 para contar que percorreu todos os caracteres dessa aresta.

O **while** das linhas 33-35 realiza a busca na árvore. Uma coisa importante é que sabemos que $P[i+1\,..\,j-1]$ está na árvore (pois já temos a árvore T_{j-1}). Então não é necessário avançar caractere por caractere das arestas que percorremos, pois sabemos que se $g + len(cn.f[P[g]].s) \leq j$ então percorrer essa string não ultrapassa a string $P[i+1\,..\,j-1]$, ou mais precisamente temos garantia que $cn.f[P[g]].s[k] = P[g+k-1]$ para todo $1 \leq k \leq len(cn.f[P[g]].s)$. Temos essa garantia pois caso contrário $P[i+1\,..\,j-1]$ não estaria na árvore. Dessa forma, podemos percorrer a aresta inteira, como é feito nas linhas 34-35. Ao final do **while**, se $g = j$ então encontramos um vértice associado a $P[i+1\,..\,j-1]$, e basta fazer $cd = len(cn.s)$ para indicarmos que estamos realmente nesse vértice, e não em um caractere da aresta incidente nele. Se $g < j$ então $P[i+1\,..\,j-1]$ está na aresta saindo de *cn* com primeiro caractere $P[g]$, e atualizamos (cn, cd) apropriadamente para indicar isso nas linhas 42-43.

Assim, a primeira parte do invariante está provada. No caso 3, o vértice interno *mid* é criado, então precisamos mostrar que seu link de sufixo é definido até a próxima extensão. O nó *mid* está associado a $P[i\,..\,j-1]$, logo seu link de sufixo precisa ser associado a $P[i+1\,..\,j-1]$, e temos que este é o nó indicado por (cn, cd). Se $cd = len(cn.s)$ (pois a busca terminou com $g = j$), então o link de sufixo de *mid* já existe e é *cn*, o que é tratado nas linhas 37-38, onde *ns* é modificado para **null** (caso já não fosse nulo). Se $cd < len(cn.s)$, então o nó que deveria ser o link de sufixo ainda não existe, mas a próxima extensão vai criá-lo ao adicionar $P[i+1\,..\,j]$, e o $(cd+1)$-ésimo caractere da aresta incidente em *cn* é diferente de $P[j]$, como mostraremos em seguida. Então a próxima extensão vai seguir o caso 3 novamente, o link de sufixo de *mid* será criado e atualizado corretamente nas linhas 25-26 pois na linha 41 guardamos em *ns* o valor *mid*, o único nó que tem o valor de *suf* indefinido.

Para mostrar que o $(cd+1)$-ésimo caractere da aresta incidente em *cn* não é $P[j]$, lembramos que executamos o caso 3 nessa extensão. Assim, T_{j-1} tinha a string $P[i\,..\,j-1]c$, para algum

caractere $c \neq P[j]$, logo, pelo Lema 4.5.1, T_{j-1} também tem a string $P[i+1\,..\,j-1]c$, e a próxima extensão será de caso 3.

□

Pelo invariante, depois da última extensão realizada, teremos $T_{|P|}$, a árvore de sufixos de P, e então o algoritmo funciona. O consumo de tempo ainda não foi analisado. A otimização dos casos limita o tempo a $\mathcal{O}(|P|^2)$, pois ambos o **for** da linha 5 e o **while** da linha 6 executam juntamente no máximo $2|P|$ extensões. O uso de links de sufixo e a "ideia" de não ter que percorrer cada caractere das arestas quando buscamos a próxima string nas linhas 33-35 reduzem o tempo para $\mathcal{O}(|P|)$, como mostraremos agora. A prova é similar à da complexidade do Aho-Corasick.

Definição. *A altura de um vértice u é o número de vértices no caminho da raiz até esse vértice. Denotamos esse valor por $h(u)$. Temos que $h(r) = 1$.*

Lema 4.5.4. *Para todo vértice u interno de uma árvore de sufixos com links de sufixo, $h(u.suf) \geq h(u) - 1$.*

Demonstração. Seja $P[a\,..\,b]$ a string associada a u. Seja $(r, u_1, \ldots, u_{h(u)-1})$ o caminho da raiz até u. Para $1 \leq i < h(u)$ a string associada a u_i é $P[a\,..\,b_i] \sqsubseteq P[a\,..\,b]$. Como o vértice u_i é interno, este tem link de sufixo para um vértice associado a $P[a+1\,..\,b_i] \sqsubseteq P[a+1\,..\,b]$. Logo $u_i.suf$ está no caminho da raiz até $u.suf$, e encontramos $h(u) - 1$ vértices distintos no caminho da raiz até $u.suf$. Portanto $h(u.suf) \geq h(u) - 1$.

Note que não podemos adicionar r a esse conjunto e afirmar que encontramos $h(u)$ vértices no caminho, pois pode ser que $u_1.suf = r$. □

Complexidade. *O algoritmo no Código 9 consome tempo $\mathcal{O}(|P|)$.*

Demonstração. Conforme a argumentação apresentada logo depois do Teorema 4.5.3, combinados, o **for** da linha 5 e o **while** da linha 6 executam $\mathcal{O}(|P|)$ extensões. Vamos provar agora que o **while** na linha 33 executa apenas $\mathcal{O}(|P|)$ iterações entre todas as extensões.

Para simplificar a notação, numere as extensões como $1, 2, \ldots, k$; sem se importar de qual iteração é cada uma. Sabemos que $k \leq 2|P|$. Seja h_i a altura do vértice cn no início da i-ésima extensão feita pelo algoritmo, e w_i o número de iterações do **while** da linha 33 nessa mesma extensão. Temos que $h_i \leq n$, para qualquer $1 \leq i \leq k+1$, e que $h_1 = 1$. Seja h_{k+1} a altura de cn depois do final da última extensão.

Na análise do Aho-Corasick, usamos que a altura do link de falha aumentava em no máximo 1 por iteração, então não poderia diminuir mais do que o número de iterações entre todas elas. O argumento aqui é análogo, mas usamos que a altura *diminui* no máximo em 2.

Vamos analisar as mudanças de altura de cn no código. As linhas 8, 16, 27, 29, 34 e 42 mudam cn e portanto também sua altura. Nas linhas 8, 34 e 42 a altura de cn aumenta em 1, na linha 27 ela diminui em 1 (pois neste caso "quebramos" a aresta e criamos o novo vértice *mid*, cujo pai é o mesmo que cn tinha no início da extensão) e nas linhas 16 e 29, pelo Lema 4.5.4, a altura diminui em *no máximo* 1 (a altura também pode aumentar).

Note que a altura de cn diminui em no máximo 2 durante uma extensão (se ambas as linhas 27 e 29 forem executadas), logo $h_{i+1} \geq h_i - 2$. Pela linha 34, em cada iteração do **while** da linha 33, a

altura de cn aumenta em 1, assim $w_i \leq h_{i+1} - (h_i - 2)$, considerando a possibilidade mais pessimista que em todas as extensões a altura diminui em exatamente 2, e todos os aumentos de altura de cn se devem à linha 34, ou seja, $h_{i+1} - (h_i - 2)$ é um limite superior para o número de iterações do **while** na i-ésima extensão. Como $h_{i+1} \geq h_i - 2$, temos que esse valor é sempre positivo. Vale que

$$\begin{aligned}
\sum_{i=1}^{k} w_i &\leq \sum_{i=1}^{k} (h_{i+1} - h_i + 2) \\
&= h_{k+1} - h_1 + 2k \quad \text{(soma telescópica)} \\
&\leq |P| - 1 + 2k \quad \text{(pois } h_{k+1} \leq |P|\text{)} \\
&\leq 5|P| - 1 \quad \text{(pois } k \leq 2|P|\text{)}.
\end{aligned}$$

Assim, o número máximo de iterações do **while** da linha 33 é $5|P| - 1$, e o algoritmo consome tempo $\mathcal{O}(|P|)$.

□

Capítulo 5

Autômato de sufixos

Ao construir e usar uma árvore de sufixos de P, temos que cada vértice representa uma substring de P, e isso é uma noção bem intuitiva, que facilita usar essa estrutura para resolver problemas com strings. No entanto, o número de vértices pode ser quadrático em $|P|$, e por isso é necessário algumas vezes comprimi-la, o que complica o código.

A estrutura de um autômato de sufixos, que veremos nesse capítulo, não envolve compressão de vértices, o que deixa o código mais simples, mas cada vértice representa algo menos intuitivo, ainda que bem útil.

5.1 Ocorrências à direita

Dizemos que i é uma ocorrência à direita de S em P se $S \sqsupseteq P[1\,..\,i]$, ou seja, S ocorre em P começando na posição $i - |S| + 1$.

Definição. *Seja $D_P(S) = \{i : S \sqsupseteq P[1\,..\,i]\}$ o conjunto das ocorrências à direita de S em P. Se P estiver claro do contexto, escrevemos $D(S)$ em vez de $D_P(S)$. Se $D_P(A) = D_P(B)$, dizemos que $A \equiv_P B$, ou seja, A e B estão na mesma classe de equivalência de ocorrências à direita. Da mesma forma, escrevemos $A \equiv B$ se P está claro do contexto.*

É fácil notar que $\equiv_P$ é uma relação de equivalência. Nesse capítulo, usaremos que AB é a concatenação das strings A e B.

Vamos provar alguns fatos sobre ocorrências à direita que ajudam a entender as classes de equivalência de $\equiv$, e as provas envolvidas na criação do algoritmo. Só trabalharemos com strings que são substrings de P, então pode-se assumir que o conjunto D das strings analisadas é não-vazio.

Lema 5.1.1. *Se $A \sqsupseteq B$ então $D(B) \subseteq D(A)$.*

Demonstração. Seja $i \in D(B)$ uma ocorrência à direita de B. Então $B \sqsupseteq P[1\,..\,i]$. Como $A \sqsupseteq B$, temos que $A \sqsupseteq P[1\,..\,i]$ e $i \in D(A)$. □

Lema 5.1.2. *Se $D(A) \subseteq D(B)$ então $A \sqsupseteq B$ ou $B \sqsupseteq A$.*

Demonstração. Se $i \in D(A)$, então $i \in D(B)$. Logo, neste caso, $A \sqsupseteq P[1\,..\,i]$ e $B \sqsupseteq P[1\,..\,i]$. Como A e B são sufixos da mesma string, $A \sqsupseteq B$ ou $B \sqsupseteq A$. □

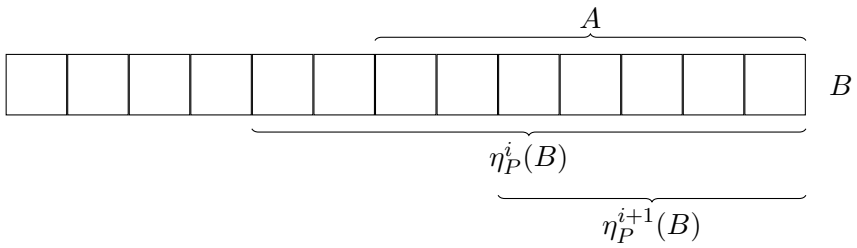


Figura 5.1: Explicação Lema 5.1.6.

Corolário 5.1.2.1. *Se $A \equiv B$ e $|A| \leq |B|$ então $A \sqsupseteq B$.*

Demonstração. Como $A \equiv B$, temos que $D(A) = D(B)$ e, pelo Lema 5.1.2, $A \sqsupseteq B$ ou $B \sqsupseteq A$. O resultado segue de $|A| \leq |B|$. □

Isso mostra que todas as strings de uma classe de equivalência estão relacionadas pois são sufixos umas das outras.

Proposição 5.1.3. *Se $A \equiv B$, e $c \in \Sigma$, então $Ac \equiv Bc$.*

Demonstração. Note que para que i esteja em $D(Ac)$ é necessário e suficiente que $i - 1 \in D(A)$ e $P[i] = c$. Como $D(A) = D(B)$, vale que $D(Ac) = D(Bc)$ e assim $Ac \equiv Bc$. □

Para todo $A \neq \varepsilon$, seja $\eta_P(A)$ a maior string $B \sqsupset A$ tal que $B \not\equiv_P A$. Note que $A \not\equiv_P \varepsilon$, logo $\eta_P(A)$ está bem definida. Dizemos que $\eta_P(A)$ é a *string de falha* de A para P. Se P está claro do contexto, escrevemos apenas $\eta(A)$.

Lema 5.1.4. *Se $A \equiv B$ então $\eta(A) = \eta(B)$.*

Demonstração. Assuma sem perda de generalidade que $|A| \leq |B|$. Então, pelo Corolário 5.1.2.1, temos que $A \sqsupseteq B$. Como $\eta(A) \sqsupset A \sqsupseteq B$ e $\eta(A) \not\equiv A \equiv B$, vale que $|\eta(B)| \geq |\eta(A)|$. Se $A \sqsupseteq \eta(B)$, pelo Lema 5.1.1, temos $D(\eta(B)) \subseteq D(A) = D(B)$, uma contradição pois $D(B) \subsetneq D(\eta(B))$ pois $\eta(B) \sqsupset B$ e $\eta(B) \not\equiv B$. Assim sendo, $\eta(B) \sqsupset A$ e, como $A \not\equiv \eta(B)$, vale que $|\eta(B)| \leq |\eta(A)|$ e o lema segue. □

Lema 5.1.5. *Se $A \neq \varepsilon$, então $\eta(A)$ é a maior string de sua classe de equivalência.*

Demonstração. Suponha que exista string B tal que $B \equiv \eta(A)$ e $|B| > |\eta(A)|$. Como $\eta(A) \sqsupset A$, o Lema 5.1.1 implica que $D(A) \subset D(\eta(A)) = D(B)$. Usando o Lema 5.1.2, temos que $A \sqsupset B$ ou $B \sqsupset A$. Se $A \sqsupset B$ então $D(B) \subset D(A)$, uma contradição. Se $B \sqsupset A$ então temos uma contradição pois $\eta(A)$ é o *maior* sufixo de A de uma classe diferente. □

Corolário 5.1.5.1. *A string $\eta_P(P)$ é o maior sufixo de P que ocorre pelo menos duas vezes em P.*

Demonstração. Note que $D(P) = \{|P|\}$, logo $|D(\eta(P))| > 1$, ou seja, $\eta(P)$ ocorre mais de uma vez em P. Pelo Lema 5.1.5, essa string é a maior de sua classe de equivalência. □

Lema 5.1.6. *Se $A \sqsupseteq B$, então existe $i \geq 0$ tal que $\eta^i(B) \equiv A$, onde usamos que $\eta^0(B) = B$ e $\eta^i(B) = \eta(\eta^{i-1}(B))$.*

Demonstração. Escolha $i \geq 0$ máximo tal que $|\eta^i(B)| \geq |A|$. Então claramente $A \sqsupseteq \eta^i(B)$. A Figura 5.1 ilustra tal situação. Se $A \not\equiv \eta^i(B)$ então pela definição de η como *maior* sufixo de outra classe de equivalência temos que $|\eta(\eta^i(B))| \geq |A|$, uma contradição à maximalidade do i escolhido. Logo $\eta^i(B) \equiv A$. □

Esses resultados são úteis para entender melhor a relação de equivalência por ocorrências à direita, e serão úteis na criação do algoritmo. A árvore de sufixos de P é uma árvore na qual cada vértice representa uma substring de P. No autômato de sufixos, criaremos um digrafo no qual cada vértice representa uma classe de equivalência de ocorrências à direita em P. Assim como na árvore, temos um vértice inicial, e arestas têm caracteres associados, mas no autômato dois caminhos podem levar ao mesmo vértice se as strings associadas a estes forem equivalentes.

5.2 Laminaridade

Seja $\mathcal{F}$ uma família de subconjuntos de um conjunto E não-vazio, ou seja, todo elemento de $\mathcal{F}$ é um subconjunto de E. Dizemos que $\mathcal{F}$ é laminar se, para quaisquer $A, B \in \mathcal{F}$, temos que $A \cap B = \emptyset$ ou $A \subseteq B$ ou $B \subseteq A$.

Teorema 5.2.1. *Seja E um conjunto não vazio, e $\mathcal{F}$ uma família de subconjuntos de E com $\emptyset \notin \mathcal{F}$. Se $\mathcal{F}$ é laminar então $|\mathcal{F}| \leq 2|E| - 1$, e se $E \notin \mathcal{F}$ então $|\mathcal{F}| \leq 2|E| - 2$.*

Demonstração. A prova é por indução em $|E|$. Se $|E| = 1$, as afirmações valem trivialmente. Suponha que $|E| > 1$ e que as afirmações valem para toda família laminar de um conjunto de tamanho menor que $|E|$. Se $E \in \mathcal{F}$, vamos provar que $\mathcal{F}' \coloneqq \mathcal{F} \setminus \{E\}$ tem tamanho no máximo $2|E| - 2$, e então $|\mathcal{F}| = |\mathcal{F}'| + 1 \leq 2|E| - 1$.

Suponha então que $E \notin \mathcal{F}$. Escolha $A \in \mathcal{F}$ maximal, ou seja, tal que não existe $B \in \mathcal{F}$ com $A \subsetneq B$. Considere $\mathcal{F}_1 = \{X \in \mathcal{F} : X \subseteq A\}$ e $\mathcal{F}_2 = \{X \in \mathcal{F} : X \cap A = \emptyset\}$. Pela maximalidade de A e porque $\mathcal{F}$ é laminar, temos que $\mathcal{F}_1 \dot\cup \mathcal{F}_2 = \mathcal{F}$. Além disso, $\mathcal{F}_1$ é laminar sobre o conjunto A e $\mathcal{F}_2$ é laminar sobre o conjunto $E \setminus A$. Ademais, $1 \leq |A| < |E|$ e $1 \leq |E \setminus A| < |E|$.

Aplicando a hipótese de indução a $\mathcal{F}_1$ e $\mathcal{F}_2$, temos que $|\mathcal{F}_1| \leq 2|A| - 1$ e $|\mathcal{F}_2| \leq 2|E \setminus A| - 1 = 2|E| - 2|A| - 1$. Assim

$$|\mathcal{F}| = |\mathcal{F}_1 \dot\cup \mathcal{F}_2| = |\mathcal{F}_1| + |\mathcal{F}_2| \leq 2|A| - 1 + 2|E| - 2|A| - 1 = 2|E| - 2.$$

□

Proposição 5.2.2. *A família de conjuntos $F(P) = \{D(A) : A$ é uma substring de $P\}$ é laminar, ou seja, se $D(A) \cap D(B) \neq \emptyset$ então $D(A) \subseteq D(B)$ ou $D(B) \subseteq D(A)$.*

Demonstração. Seja $i \in D(A) \cap D(B)$. Então, pela definição, $A \sqsupseteq P[1..i]$ e $B \sqsupseteq P[1..i]$. Logo $A \sqsupseteq B$ ou $B \sqsupseteq A$ e pelo Lema 5.1.1 segue que $D(B) \subseteq D(A)$ ou $D(A) \subseteq D(B)$. □

Corolário 5.2.2.1. *O tamanho de $F(P)$ não excede $2|P|$.*

Demonstração. Como $D(A) \subseteq \{1, 2, \ldots, |P|\}$ para todo $D(A) \in F(P)$ tal que $A \neq \varepsilon$, pelo Teorema 5.2.1 vale que $|F(P) \setminus \{D(\varepsilon)\}| \leq 2|P| - 1$ e então $|F(P)| \leq 2|P|$. □

Isso mostra que o número de classes de equivalência de $\equiv_P$ é linear em $|P|$, mesmo que o número de substrings distintas de P possa ser quadrático em $|P|$.

5.3 Refinamento das classes de equivalência

Construiremos o autômato de forma online, similarmente à árvore de sufixos, prefixo por prefixo. Por isso precisamos estudar como a relação de equivalência $\equiv_P$ muda em relação a $\equiv_{Pc}$, onde c é um caractere adicionado ao fim da string P.

Ao adicionar um caractere ao final de P, apenas uma nova ocorrência à direita é possível: a ocorrência na posição $|P|+1$. Logo é claro que

$$D_{Pc}(A) = \begin{cases} D_P(A) \cup \{|P|+1\} & \text{se } A \sqsupseteq Pc \\ D_P(A) & \text{caso contrário.} \end{cases} \tag{5.1}$$

Uma consequência direta disso é a seguinte.

Proposição 5.3.1. *Se $A \equiv_{Pc} B$ então $A \equiv_P B$, ou seja, $\equiv_{Pc}$ é um refinamento de $\equiv_P$.*

Demonstração. Se $A \sqsupseteq Pc$, então $D_{Pc}(A) = D_P(A) \cup \{|P|+1\}$, e assim temos que $|P|+1 \in D_{Pc}(A) = D_{Pc}(B)$. Logo, $B \sqsupseteq Pc$ e $D_{Pc}(B) = D_P(B) \cup \{|P|+1\}$. Isso implica que $D_P(A) = D_P(B)$ e então $A \equiv_P B$.

Analogamente, se $A \not\sqsupseteq Pc$, então $D_{Pc}(A) = D_P(A)$, e $|P|+1 \notin D_{Pc}(A) = D_{Pc}(B)$, logo $D_{Pc}(B) = D_P(B)$ e assim $A \equiv_P B$. □

O lema mostra que, quando adicionamos um caractere ao final de P, as classes de equivalência ou se quebram em classes menores ou se mantém as mesmas, mas nunca se juntam com outras.

Além disso, uma classe nova surge (a classe de equivalência de Pc), e apenas uma classe de equivalência se quebra em classes menores.

Proposição 5.3.2. *Seja $X = \eta_{Pc}(Pc)$. Se $A \sqsupseteq Pc$ e $|A| > |X|$ então $A \equiv_{Pc} Pc$.*

Demonstração. Temos que $D_{Pc}(Pc) = \{|P|+1\} = D_{Pc}(A)$, pois pelo Corolário 5.1.5.1 X é o *maior* sufixo de Pc que ocorre em P. Logo A não ocorre em P. □

Teorema 5.3.3. *Seja $X = \eta_{Pc}(Pc)$. Se $A \equiv_P B$ e $A \not\equiv_P X$, onde A e B são substrings de P, então $A \equiv_{Pc} B$.*

Demonstração. Vamos mostrar que $A \sqsupseteq Pc$ é equivalente a $B \sqsupseteq Pc$. Suponha que $A \sqsupseteq Pc$. Então $A \sqsupseteq X$, pois X é o maior sufixo de Pc que ocorre em P pelo Corolário 5.1.5.1. Pelo Lema 5.1.6, existe $i \geq 0$ tal que $Y = \eta_P^i(X) \equiv_P A$. Como $A \not\equiv_P X$, temos que $i > 0$ e pelo Lema 5.1.5 vale que Y é a maior string de sua classe de equivalência. Assim, pelo Corolário 5.1.2.1, temos que $B \sqsupseteq Y \sqsupset X \sqsupset Pc$.

Como não usamos nenhuma característica especial de A acima, de maneira semelhante podemos provar que $B \sqsupseteq Pc$ implica $A \sqsupseteq Pc$. Como $A \sqsupseteq Pc$ é equivalente a $B \sqsupseteq Pc$, temos por (5.1) que $A \equiv_{Pc} B$. □

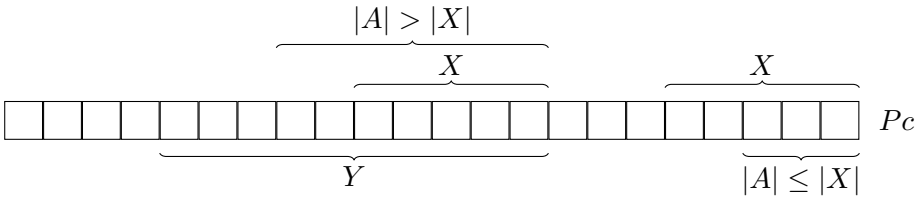


Figura 5.2: Explicação Teorema 5.3.4.

Esse teorema mostra que, entre as classes de equivalência das substrings de P, apenas a classe de X pode se dividir, todas as outras continuam as mesmas.

Teorema 5.3.4. *Seja $X = \eta_{Pc}(Pc)$, e Y uma string de comprimento máximo tal que $X \equiv_P Y$. Seja A uma substring de P tal que $A \equiv_P X$, então*

$$A \equiv_{Pc} \begin{cases} X & \text{se } |A| \leq |X| \\ Y & \text{caso contrário.} \end{cases}$$

Demonstração. Pelo Lema 5.1.5, temos que X deve ser o maior de sua classe de equivalência em relação a Pc, mas não a P. Por isso pode ser que $Y \neq X$. Nesse caso, $X \sqsupset Y$ pelo Corolário 5.1.2.1. Isso pode ocorrer como na Figura 5.2. Note que $X \sqsupset Y$ mas $Y \not\sqsupseteq Pc$.

Por (5.1), a classe de equivalência de X se divide em no máximo duas classes: a dos elementos que são sufixos de Pc e a dos elementos que não são sufixos de Pc. Se $|A| \leq |X|$, usando o Corolário 5.1.2.1, temos que $A \sqsupseteq X \sqsupseteq Pc$. Logo A continua na mesma classe que X. Se $|A| > |X|$, então $X \sqsupset A$ e como X é o *maior* sufixo de Pc que ocorre em P, pelo Corolário 5.1.5.1, temos que $A \not\sqsupseteq Pc$ e também $Y \not\sqsupseteq Pc$. Logo ambos A e Y continuam na mesma classe de equivalência de $\equiv_{Pc}$, diferente da classe de X. □

5.4 Representação

A teoria desenvolvida já dá uma ideia de como escrever um algoritmo para construir o autômato. Queremos construir um DAG (Grafo Dirigido Acíclico) em que cada nó represente uma classe de equivalência de $\equiv_P$ (ou seja, um conjunto de ocorrências à direita), e uma aresta indique, ao adicionar um caractere ao final de uma string da classe de equivalência do nó cabeça da aresta, para qual classe de equivalência esta string muda. A Proposição 5.1.3 mostra que a classe de equivalência ao adicionar um caractere não depende da string escolhida.

O DAG terá um nó inicial (chamaremos de raiz) que indica o nó relativo à classe de equivalência de ε. Um caminho a partir da raiz representa uma substring de P, e note que vários caminhos podem levar ao mesmo nó.

A Seção 5.3 mostra como atualizar esses nós ao adicionar um caractere a P. A Proposição 5.3.2 mostra quais novas substrings existem, que tem a mesma classe que a string Pc. O Teorema 5.3.3 prova que apenas a classe de $X = \eta_{Pc}(Pc)$ muda, podendo se particionar em duas. Esse particionamento é então detalhado no Teorema 5.3.4.

Seja u uma classe de equivalência de $\equiv_P$. Usaremos que u é um nó do autômato, e que $u.f$ é a lista de adjacências de u, ou seja, guarda todas as arestas que saem de u, indexadas por seu caractere.

Assim como anteriormente, $u.f$ é implementada usando um vetor de $|\Sigma|$ posições. Diferentemente da árvore de sufixos, cada aresta representa apenas um caractere, logo a implementação dessa lista é mais simples. Usamos que $S(u)$ é a maior string da classe u.

Usaremos que $u.\eta$ é a classe de equivalência da string de falha de alguma string da classe u. Note que, pelo Lema 5.1.4, esse valor independe da string escolhida na classe u, logo está bem definido. Usaremos que $u.L$ é o tamanho da maior string em u, ou seja, é $|S(u)|$. Além disso, usamos **new** $node(x)$ para representar a criação de um nó com campo L inicializado com x, lista de adjacências vazia e campo η nulo.

O autômato será construído adicionando um caractere por vez ao final da string. Assim, ao construir o autômato de Pc, é necessário atualizar as falhas η_P para η_{Pc}, e o seguinte lema ajuda nisso.

Lema 5.4.1. *Se $A \sqsupseteq P$ e $\eta_{Pc}(Ac) \neq \varepsilon$, então $\eta_{Pc}(Ac) = \eta_P^i(A)c$ para algum $i > 0$ e, para todo $j < i$, vale que $\eta_P^j(A)c \equiv_P Ac$.*

Demonstração. Seja $Bc = \eta_{Pc}(Ac)$. Pela definição de η, temos que $B \sqsupseteq A$. Suponha que exista $Z \equiv_P B$ tal que $B \sqsupset Z$. Então, pela Proposição 5.1.3, vale que $Bc \equiv_P Zc$, e como ambos B e Z são sufixos de P, vale que $\eta_{Pc}(Ac) \equiv_{Pc} Zc$ e $|Zc| > |\eta_{Pc}(Ac)|$, uma contradição pelo Lema 5.1.5. Logo $B = \eta_P^i(A)$ para algum $i > 0$ (caso contrário, η "pularia" uma classe de equivalência), e assim $\eta_{Pc}(Ac) = \eta_P^i(A)c$.

Pela definição de η, para todo $j < i$, vale $\eta_P^j(A)c \equiv_{Pc} Ac$ e, usando a Proposição 5.3.1, temos $\eta_P^j(A)c \equiv_P Ac$. □

5.5 Implementação

O algoritmo no Código 10 constrói o autômato de sufixos para uma string P, de acordo com o que foi discutido. A Figura 5.3 ilustra como a construção funciona para a string `banana`.

Invariante. *No início da i-ésima iteração do **for** da linha 3, root é a raiz do autômato de $P[1\,..\,i-1]$ e last é o nó que representa a classe de $P[1\,..\,i-1]$ nesse autômato.*

Demonstração. Na primeira iteração do **for**, o autômato está corretamente calculado, pois o autômato de uma string vazia é apenas um nó (associado à classe da string ε), e a linha 2 inicializa corretamente os campos *root* e *last*.

No começo de uma iteração, temos o autômato de $P[1\,..\,i-1]$ e queremos calcular o autômato de $P[1\,..\,i]$, para que o invariante continue valendo. Para continuar com a mesma notação de antes, vamos usar que temos o autômato de $Q = P[1\,..\,i-1]$ e queremos calcular o autômato de Qc, onde $c = P[i]$.

Uma nova classe de equivalência sempre se forma, a de Qc, e de acordo com a Proposição 5.3.2, todos os sufixos de Qc maiores que $X = \eta_{Qc}(Qc)$ estão nessa classe de equivalência. A variável *last* é atualizada para o novo nó que representará essa classe. A variável p passa a guardar o valor antigo de *last*, e será utilizada para iterar pelas classes dos sufixos de Q.

O **while** nas linhas 6-8 itera pelas classes de equivalência dos sufixos de Q enquanto $p.f[c] =$ **null**, ou seja, enquanto $S(p)c \equiv_Q Qc$ (pois $D_Q(Qc) = \emptyset$). Vale então, pelo Lema 5.4.1, que após esse

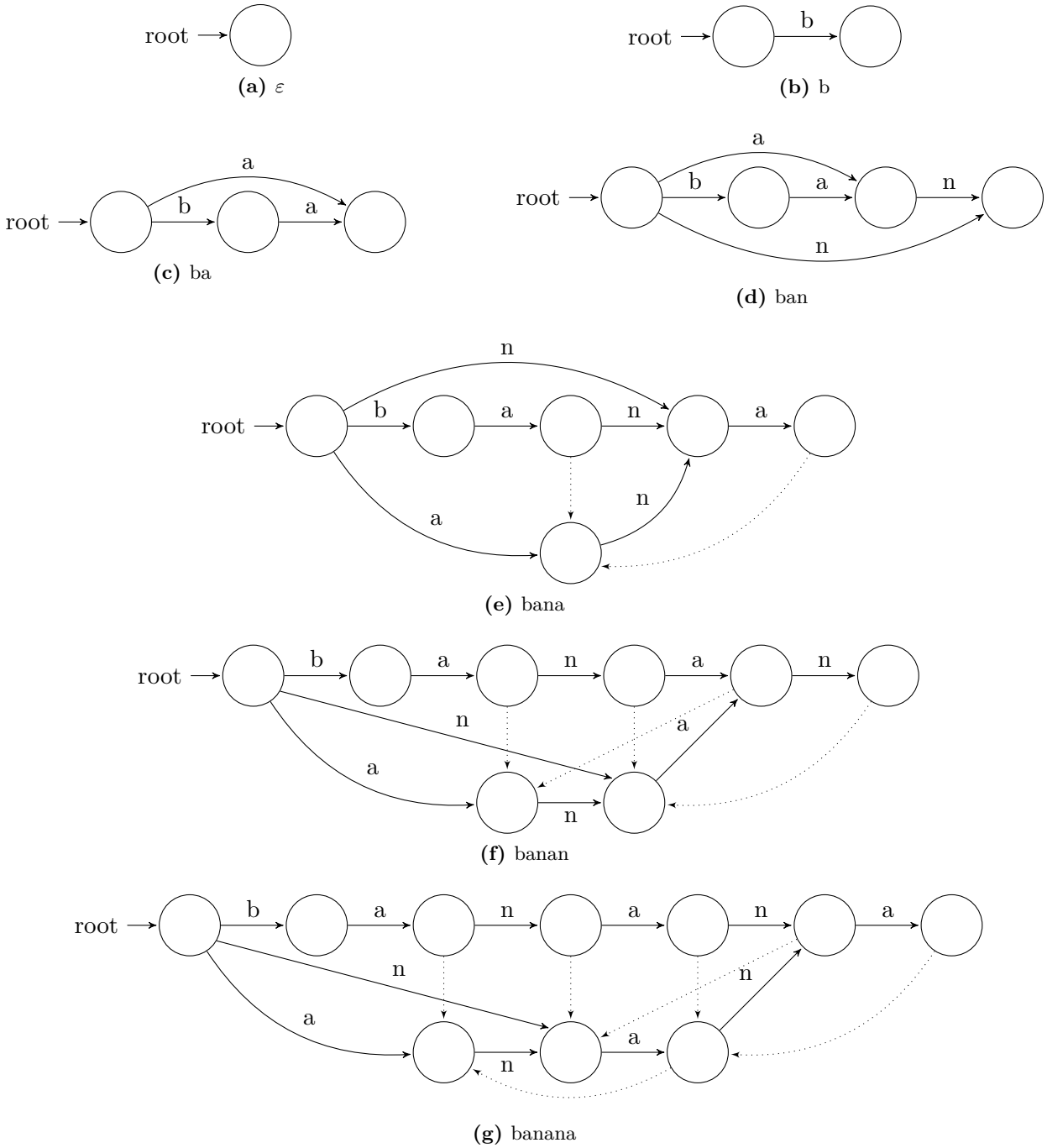


Figura 5.3: Construção do autômato de sufixos de `banana`, falhas desenhadas pontilhadas. As falhas para a raiz não estão desenhadas para não obstruir a imagem.

Código 10 Autômato de sufixos em tempo linear

```
1:  function BuildSuffixAutomaton(P)
2:      last = root = new node(0)
3:      for i = 1 to |P| :
4:          p = last
5:          last = new node(i)                          ▷ classe da string P[1..i]
6:          while p ≠ null and p.f[P[i]] = null :
7:              p.f[P[i]] = last
8:              p = p.η
9:          if p = null :
10:             last.η = root
11:         else
12:             y = p.f[P[i]]                           ▷ X é a string S(p)P[i]
13:             if y.L = p.L + 1 :                      ▷ se X = S(y)
14:                 last.η = y
15:             else                                    ▷ a classe de X se divide em duas
16:                 x = new node(p.L + 1)
17:                 x.f = y.f
18:                 x.η = y.η
19:                 y.η = last.η = x
20:                 while p ≠ null and p.f[P[i]] = y :
21:                     p.f[P[i]] = x                   ▷ Trocando adjacência de y para x
22:                     p = p.η
23:     return root
```

while, se temos $p \neq$ **null** então $S(p)c = X$, e assim o nó $p.f[c]$ é a classe de equivalência de X. Se $p =$ **null**, então o caractere c não ocorre em Q e a linha 10 corretamente atualiza a falha η de *last* para indicar a raiz, pois nesse caso $\eta_{Qc}(Qc) = \varepsilon$. Note que a linha 7 atualiza a lista de adjacências dos nós acessados por p, e está correta pela Proposição 5.3.2.

Se a linha 12 é atingida, temos que y é a classe de equivalência da string X. Então, pelo Teorema 5.3.4, essa classe pode se quebrar em duas classes de equivalência. As linhas 13-14 tratam o caso em que $X = S(y)$, ou seja, no teorema temos $X = Y$ e a classe de equivalência não se separa. As linhas 16-22 tratam o caso em que $S(y) = Y \neq X$. Nesse caso, é necessário quebrar a classe em duas, e o novo nó criado x vai ser a classe das strings em y com tamanho que não excede $|X| = p.L + 1$. A linha 17 copia a lista de adjacências de y para x, já que as adjacências continuam iguais, e a linha seguinte copia a falha η de y para x. A linha 19 atualiza as falhas de *last* e y, pois ambas devem ser x. De fato, $last.\eta = x$ pois a string $X = \eta_{Qc}(Qc)$ está em x e $y.\eta = x$ pois $X \sqsupset Y$ e tem tamanho máximo, já que antes Y e X estavam na mesma classe.

Resta atualizar todo nó u tal que $u.f[c] = y$ e $|S(u)| \leq |X|$ para $u.f[c] = x$, então pelo Teorema 5.3.4 essas classes estarão corretas. Note que, para tais nós u, teremos $S(u)c \sqsupseteq X = S(p)c$, e assim podemos acessá-los pelas falhas de p. O **while** das linhas 20-22 atualiza as listas de adjacências necessárias, pois iteramos enquanto $p.f[c] = y$, ou seja, enquanto $S(p)c \equiv_Q X$.

Pelo Teorema 5.3.3, todas as outras classes de equivalências continuam as mesmas, e assim o autômato de $Qc = P[1..i]$ é corretamente calculado, mantendo o invariante. □

Pelo invariante, temos que ao final do algoritmo o autômato de sufixos de P está corretamente

calculado. Provar a complexidade, contudo, não é tão simples. O Corolário 5.2.2.1 mostra que o número de nós do autômato é $\mathcal{O}(|P|)$, mas ainda temos que o número de arestas pode ser $\Theta(|P||\Sigma|)$, se de cada vértice sair uma aresta para cada vértice, por exemplo. Esse, porém, não é caso, como mostraremos abaixo.

Dizemos que um nó é final se ele representa uma classe de equivalência de um sufixo de P. Um caminho da raiz à um nó final indica um sufixo de P. A partir de um nó não-final, é sempre possível encontrar um caminho a um nó final, pois é sempre possível completar uma substring de P para que esta se torne um sufixo. Essas noções serão úteis na prova da proposição abaixo.

Proposição 5.5.1. *O número de arestas do autômato de sufixos de P não excede $3|P| - 1$.*

Demonstração. Como a partir da raiz é possível atingir qualquer vértice, seja T uma árvore geradora qualquer do autômato.

Para cada aresta $e = uv$ que não pertence a T, considere a string AcB, onde A é a string associada ao caminho da raiz a u na árvore T, c é o caractere associado à aresta e, e B é a string associada ao caminho de v a algum vértice final (tal caminho sempre existe pois é possível completar uma substring para um sufixo).

Todas essas strings são distintas, já que cada uma tem a primeira aresta que não está em T diferente. Além disso, como acabam em um vértice final, temos que $AcB \sqsupseteq P$, para toda $e \notin T$. Logo, existem no máximo $|P|$ arestas fora da árvore, pois P tem $|P|$ sufixos. Pelo Corolário 5.2.2.1, temos que o número de arestas em T é no máximo $2|P| - 1$, e assim o número de arestas do autômato é no máximo $3|P| - 1$. □

Complexidade. *O algoritmo no Código 10 consome tempo $\Theta(|P||\Sigma|)$.*

Demonstração. Como o **for** da linha 3 tem exatamente $|P|$ iterações, todas as operações neste que consomem tempo constante (atribuições, somas, etc.) vão no total consumir tempo $\mathcal{O}(|P|)$. Se consideramos que as operações **new** *node* consomem tempo $\Theta(|\Sigma|)$, então no total elas consumirão tempo $\Theta(|P||\Sigma|)$. Resta analisar os **while**s das linhas 6 e 20, e a instrução da linha 17, pois esta não consome tempo constante.

A instrução de copiar a lista de adjacências na linha 17 precisa copiar cada aresta de uma lista de adjacências para outra. Como estamos considerando a implementação dessa lista usando um vetor, é necessário copiar todas as $|\Sigma|$ posições de um vetor para outro, então no total isso consome tempo $\mathcal{O}(|P||\Sigma|)$.

Como já foi provado pelo Corolário 5.2.2.1 e pela Proposição 5.5.1, o número de vértices e arestas do autômato final é $\mathcal{O}(|P|)$, e note que o número de vértices e arestas nunca diminui durante o algoritmo. Logo, as operações que ocorrem apenas uma vez por criação de nó ou aresta levam tempo linear. Isso mostra que o **while** da linha 6 consome tempo linear, pois cada iteração deste cria uma aresta. Resta então provar a complexidade do **while** das linhas 20-22, pois nessas linhas estamos "trocando" arestas, então isso potencialmente poderia ocorrer muitas vezes.

Considere novamente que na i-ésima iteração temos o autômato de sufixos da string $Q = P[1\,..\,i-1]$ e iremos adicionar o caractere $c = P[i]$. Na linha 12, temos que y é a classe de equivalência de $X = \eta_{Qc}(Qc)$ na string Q, então considere o Lema 5.4.1, onde $Ac = X$. Temos que p é a classe de equivalência de A. Note que o **while** itera pelas falhas de p enquanto $p.f[c] = y$,

ou seja, enquanto $S(p) \equiv_Q X$. Pelo Lema 5.4.1, temos que, quando esse laço acaba, $p.f[c]$ é a classe de equivalência de $\eta_{Qc}(X)$ na string Qc, ou seja, a classe de $\eta^2_{Qc}(Qc)$. Note que $\eta^2_{Qc}(Qc) = S(p)c$, ou seja, $|\eta^2_{Qc}(Qc)| = p.L + 1$.

Agora vamos novamente usar um argumento similar ao da prova da complexidade do Aho-Corasick e da árvore de sufixos. Vamos observar o valor de $p.L$ durante o algoritmo, ou seja, o tamanho da maior string de p. Note que a operação $p = p.\eta$ sempre diminui $p.L$ em pelo menos 1. Em cada iteração, p é inicializado como *last*, ou seja, inicialmente $p.L = i - 1$. Perceba que as condições dos **while**s das linhas 6 e 20 são sempre satisfeitas pelo menos uma vez (de fato, poderíamos ter usado **do-while**), logo ao final da iteração temos $p.L \leq |\eta^2_Q(Q)|$.

Usando o que foi discutido anteriormente, temos $|\eta^2_{Qc}(Qc)| = p.L + 1 \leq |\eta^2_Q(Q)| + 1$. Assim vale que o valor de $|\eta^2_{P[1\,..\,i]}(P[1\,..\,i])|$ sempre aumenta em no máximo 1 entre iterações, e cada iteração do **while** da linha 6 ou da linha 20 diminui esse valor em pelo menos 1. Seja w_i o número de iterações além da primeira dos dois **while**s. Vale que $w_i \leq (|\eta^2_{P[1\,..\,i-1]}(P[1\,..\,i-1])| + 1) - |\eta^2_{P[1\,..\,i]}(P[1\,..\,i])|$. Para que essa notação funcione mesmo quando $i = 1$, vamos usar que $\eta_\varepsilon(\varepsilon) = \varepsilon$. Temos então que

$$\begin{aligned}
\sum_{i=1}^{|P|} w_i &\leq \sum_{i=1}^{|P|} (|\eta^2_{P[1\,..\,i-1]}(P[1\,..\,i-1])| - |\eta^2_{P[1\,..\,i]}(P[1\,..\,i])| + 1) \\
&= |\eta^2_\varepsilon(\varepsilon)| - |\eta^2_P(P)| + |P| \quad \text{(soma telescópica)} \\
&\leq |P| \quad \text{(pois } |\eta^2_\varepsilon(\varepsilon)| = 0 \text{ e } \ |\eta^2_P(P)| \geq 0).
\end{aligned}$$

Isso prova que os **while**s levam tempo linear e assim o algoritmo tem complexidade de tempo $\Theta(|P||\Sigma|)$.

□

É fácil ver que o espaço da estrutura criada também é $\Theta(|P||\Sigma|)$. Usando a implementação de listas de adjacências como ABBBs, usando o caractere de uma aresta como chave, e sua ponta final como valor, é possível provar de forma muito semelhante que o algoritmo consome tempo $\Theta(|P| \log |\Sigma|)$ e espaço $\Theta(|P|)$.

5.6 Uso

Com o autômato de P construído, é simples saber se uma string S é substring de P. Basta verificar se existe caminho $S[1], S[2], \ldots, S[|S|]$ no autômato, a partir da raiz.

Contar o número de ocorrências ou enumerá-las não é tão simples. É necessário descobrir quais são os nós finais do nosso autômato.

Proposição 5.6.1. *Um nó do autômato de P é final se e somente se pertence ao conjunto* $\textsc{Path}(last) = \{last, last.\eta, last.\eta.\eta, \ldots, root\}$.

Demonstração. O nó *last* claramente é final, pois é a classe de equivalência de P. Assim, como $S(u.\eta) \sqsupset S(u)$ para qualquer $u \neq root$, pela definição de η, temos que qualquer elemento do conjunto $\textsc{Path}(last)$ é um nó final. Por outro lado, se $X \sqsupseteq P$, pelo Lema 5.1.6, existe $i \geq 0$ tal que $\eta^i(P) \equiv X$, e portanto a classe de equivalência de X está em $\textsc{Path}(last)$. □

Portanto, se adicionarmos um campo booleano *final* à nossa implementação de nós, que é inicializado com **false**, e indica se um nó é final ou não, o seguinte código pode ser adicionado ao final da criação do autômato de sufixos para marcar os nós finais corretamente.

```
1: p = last
2: while p ≠ null :
3:     p.final = true
4:     p = p.η
```

Agora, para contar o número de ocorrências de alguma string S em P, percorremos o caminho $S[1], \ldots, S[|S|]$ no autômato de sufixos de P. Seja v o nó alcançado. O número de ocorrências de S em P é o número de caminhos distintos de v para um nó final, pois este é o número de formas de completar a string S para ser um sufixo de P. Isso é similar a como, na árvore de sufixos, o número de ocorrências de uma substring representada por um nó é o número de folhas na subárvore deste.

Como o autômato de sufixos é um DAG, ele tem uma ordenação topológica, e por isso calcular o número de caminhos a partir de cada vértice para algum nó final é simples. Para calcular uma ordenação topológica de um DAG, basta realizar uma DFS e armazenar os vértices por ordem inversa de finalização da DFS, como discutido por Cormen et al. [Cor+09, Seção 22.4]. Dado um vetor *top* representando uma ordenação topológica do autômato, ou seja, se $top[i]top[j]$ é uma aresta, então $i < j$, é possível calcular o número de caminhos de u para algum vértice final ($\textsc{Count}(u)$) usando a seguinte recorrência

$$\textsc{Count}(u) = \begin{cases} \sum\limits_{v \in u.f} \textsc{Count}(v) & \text{se } u \text{ não é final} \\ 1 + \sum\limits_{v \in u.f} \textsc{Count}(v) & \text{se } u \text{ é final.} \end{cases}$$

Ou seja, o número de caminhos de u para algum vértice final é o número de caminhos de seus vizinhos para algum vértice final (tratando o caso quando esses caminhos tem comprimento maior que 0), somado de 1 se u for final (tratando o caso quando o comprimento é 0). A recorrência está correta pois o digrafo é acíclico.

O seguinte código determina o valor de $\textsc{Count}$ para cada vértice u em $C[u]$.

```
1: for i = |top| down to 1 :
2:     C[top[i]] = 0
3:     for v ∈ top[i].f :
4:         C[top[i]] += C[v]
5:     if top[i].final :
6:         C[top[i]] += 1
```

Se temos como invariante que, ao analisar o vértice $top[i]$, todos os valores de C para vértices $top[j]$, com $j > i$, já estão calculados, então, como *top* é uma ordenação topológica, todos os vértices adjacentes a $top[i]$ tem o valor correto, logo o valor $C[top[i]]$ é calculado corretamente. A base é trivial quando $i = |top|$.

Logo, este código funciona e consome tempo proporcional ao número de vértices mais arestas

do autômato, que é linear em $|S|$, pelo Corolário 5.2.2.1 e pela Proposição 5.5.1.

Parte Subjetiva

O processo, desafios e frustrações

Escolhi o tema pois durante a minha graduação me identifiquei com a área de teoria, porém com um foco em implementação. Isso se deve em grande parte por praticar programação competitiva, participando da Maratona de Programação. O tema em strings, e não grafos, programação dinâmica ou algum outro tópico bem visto em programação competitiva se deve ao fato que eu tinha pouca familiaridade com strings, e quis então aprender e me especializar nessa área.

Decidi estudar algoritmos complicados como árvore e autômato de sufixos, que poucas pessoas mesmo em programação competitiva conhecem, pelo desafio de conseguir implementar uma estrutura complicada de forma eficiente. E foi um grande desafio, pois muitas vezes, ao estudar esses temas, eu encontrava grande dificuldade em entender os conceitos, e não conseguia continuar, parando por dias, ou até semanas, até voltar ao assunto.

Não tinha passado muitas vezes por isso, e aprendi que esse é um passo importante para aprender um tópico difícil. Mesmo desistindo do assunto por algum tempo, sempre voltava ao assunto com uma base mais forte, e conseguia, depois de duas, três, ou mais tentativas, finalmente entender o tópico em suas entranhas, e saber por que cada parte funciona. Esse sentimento é muito gratificante, depois de semanas de frustração.

Graduação e o trabalho de formatura

As matérias «Introdução à Teoria dos Grafos» e «Otimização Combinatória» me ensinaram a provar afirmações e pensar formalmente, com atenção a afirmações vazias ou confusões comuns nas provas.

As matérias «Algoritmos em Grafos» e «Análise de Algoritmos» também me ensinaram essas questões, em menor grau, mas com grande atenção à implementação e como provar corretude e complexidade de código.

Praticar programação competitiva, e também a matéria «Desafios de Programação», me fizeram entender claramente o meu código, e conseguir implementar ideias complicadas de forma muito mais simples do que eu já teria imaginado.

Futuro

Nessa área, tenho interesse em explorar um pouco mais as estruturas desenvolvidas no texto. A árvore de sufixos, por exemplo, é simples de ser implementada para armazenar múltiplas strings, e já resolvi diversos problemas com esta. O autômato de sufixos, entretanto, não consegui desenvolver para múltiplas strings, e nem resolver algum problema de programação competitiva que não havia resolvido com árvore de sufixos.

Outro tópico interessante de se estudar são as similaridades entre árvore e autômato de sufixos, pois é possível construir um a partir do outro de forma muito mais simples que construir do início. Acredito que estudar isso daria um entendimento maior das duas estruturas.

Agradecimentos

Gostaria de agradecer a prof. Cristina, por aceitar minhas ideias de tema e me apoiar no estudo dessas estruturas, sempre tendo tempo para rabiscar meus rascunhos e me ajudando a escrever um texto melhor!

Agradeço também a todos meus amigos que praticam programação competitiva comigo, em especial no grupo MaratonIME, pois a motivação destes me fez começar a estudar bastante nessa área e me mantém focado nisso até hoje!

Bibliografia

[AC75] Alfred V. Aho e Margaret J. Corasick. ≪Efficient string matching: an aid to bibliographic search≫. Em: *Commun. ACM* 18 (6) (jun. de 1975), pp. 333–340. ISSN: 0001-0782. DOI: 10.1145/360825.360855.

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 9780262033848.

[CHL07] M. Crochemore, C. Hancart e T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. ISBN: 9781139463850.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. EBL-Schweitzer. Cambridge University Press, 1997. ISBN: 9780521585194.

[Man75] Glenn Manacher. ≪A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string≫. Em: *J. ACM* 22 (3) (jul. de 1975), pp. 346–351. ISSN: 0004-5411. DOI: 10.1145/321892.321896. URL: http://doi.acm.org/10.1145/321892.321896.

[Ukk95] Esko Ukkonen. ≪On-line construction of suffix trees≫. Em: *Algorithmica* 14 (3) (1995), pp. 249–260. ISSN: 1432-0541. DOI: 10.1007/BF01206331.

[Wei73] Peter Weiner. ≪Linear pattern matching algorithms≫. Em: *14th Symposium on Switching and Automata Theory (SWAT)* (1973). ISSN: 0272-4847. DOI: 10.1109%2FSWAT.1973.13.