String matching y Árboles de sufijos
(Estructuras de datos avanzadas)

Algoritmia para problemas difíciles
30-11-17
Elvira Mayordomo

Contenido

- **El problema de string matching**
- Árboles de sufijos
- Vectores de sufijos

Referencias

- El capítulo 8.1 (8.1.3) de Steven S. Skiena. The Algorithm Design Manual. Springer 2008.
- H.J. Böckenhauer, D. Bongartz: *Algorithmic aspects of bioinformatics*. Springer, 2007.
- D. Gusfield: *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: *Introduction to Algorithms* (3rd ed.). MIT Press, 2009.

El problema del string matching

- Consiste en encontrar un string (corto), el *patrón*, como substring de un string (largo), el *texto*

Dominios de aplicación

- Inumerables: procesado de información (text mining), robótica (visión, etc), bioinformática, etc
- Por ejemplo en bioinformática lo más frecuente es buscar un fragmento nuevo de DNA (un gen) en una colección de secuencias
- Se puede permitir un cierto error, pero el pattern matching exacto es un bloque necesario

Enunciado del problema …

- Entrada: Dos strings $t = t_1 \ldots t_n$, $p = p_1 \ldots p_m$ sobre Σ
- Salida: El conjunto de posiciones de t donde aparece p, es decir,

  $I \subseteq \{1, \ldots, \quad n-m+1\}$

  tales que $i \in I$ sii $t_i \ldots t_{i+m-1} = p$

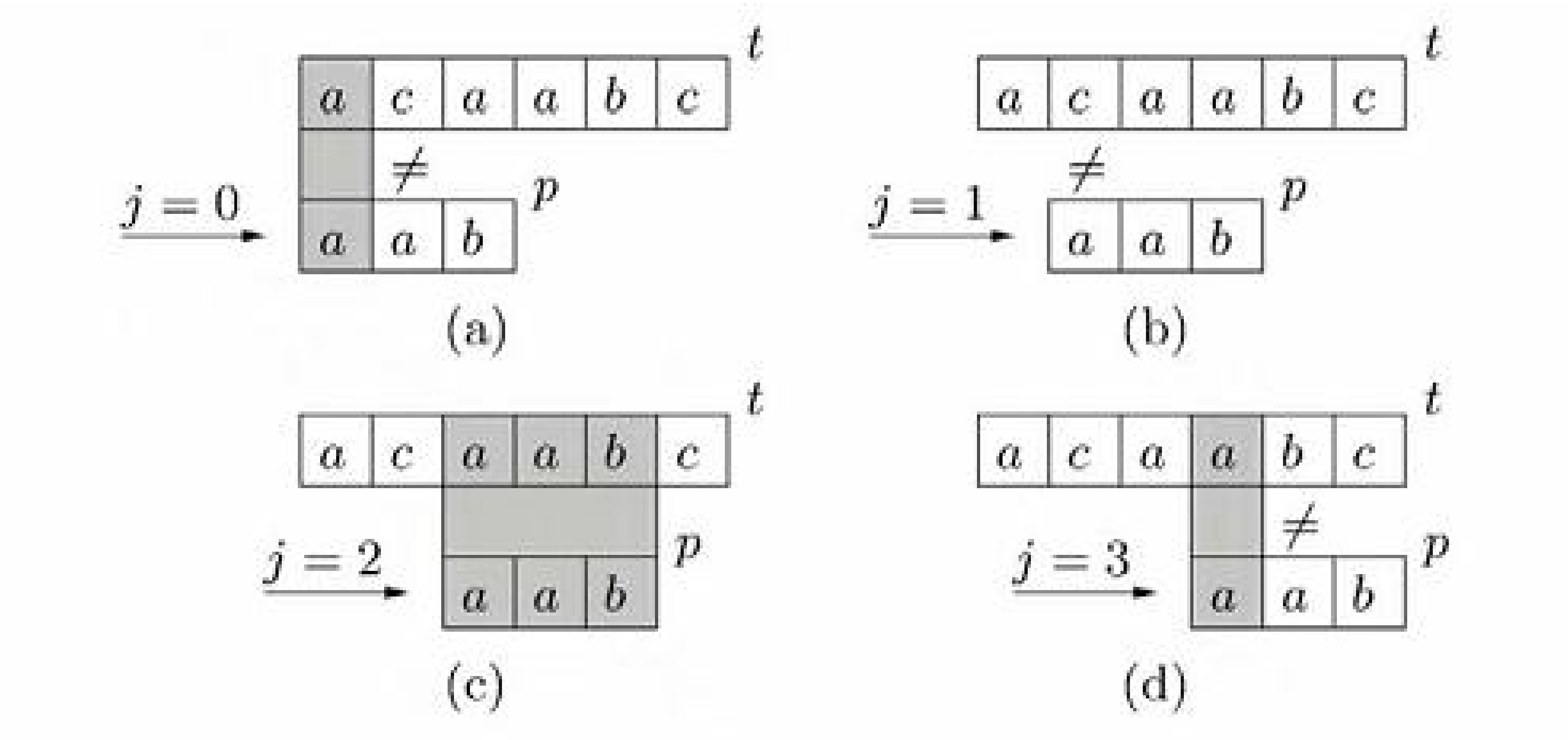
Algoritmo inocente …

StringMatching(patrón $p=p_1. . .p_m$, texto $t=t_1. . .t_n$)

```
I:=vacío
for j:=0 to n-m do
        i:=1
        while p_i = t_{j+i} and i ≤ m do
                i:=i+1
        if i=m+1 then {p_1 … p_m = t_{j+1} … t_{j+m}}
                añadir(I,j+1}
```

Resultado I (El conjunto I de posiciones, donde empieza una ocurrencia de p como substring de t)

Ejemplo …



t
a c a a b c
≠
p
j = 0
a a b
(a)
t
a c a a b c
≠
p
j = 1
a a b
(b)
t
a c a a b c
p
j = 2
a a b
(c)
t
a c a a b c
≠
p
j = 3
a a b
(d)

Algoritmo inocente …

- ¿Complejidad?
- ¿Casos peores?

- Buscamos mejorar esto explotando la estructura del patrón (o la del texto)

t= ababb   p=abb

Resto de métodos

- Preprocesar el patrón
- Preprocesar el texto (aquí entran los árboles de sufijos)

Distintos casos

- Dependiendo del caso se puede buscar el mismo patrón en muchos textos o distintos patrones en el mismo texto
- Por ejemplo buscar la misma mutación en muchos pacientes o todas las mutaciones que tiene un paciente
- Veremos primero algoritmos para el mismo patrón en muchos textos
- Luego árboles de sufijos para buscar distintos patrones en el mismo texto

Métodos clásicos …

- Vamos a ver por encima dos métodos clásicos: string matching automata y Boyer-Moore
- Los dos son algoritmos clásicos interesantes que por lo menos hay que conocer …

Preprocesar el patrón: String matching automata

- Una solución en la que una vez preprocesado p en tiempo $O(|p| \cdot |\Sigma|)$ resolvemos el problema en una sola pasada de t
- Usa autómatas finitos …

Autómata finito (DFA)

Un DFA es una 5-tupla $(Q, \Sigma, q_0, \delta, F)$ donde:

1) Q : conjunto de estados

2) Σ : alfabeto de entrada

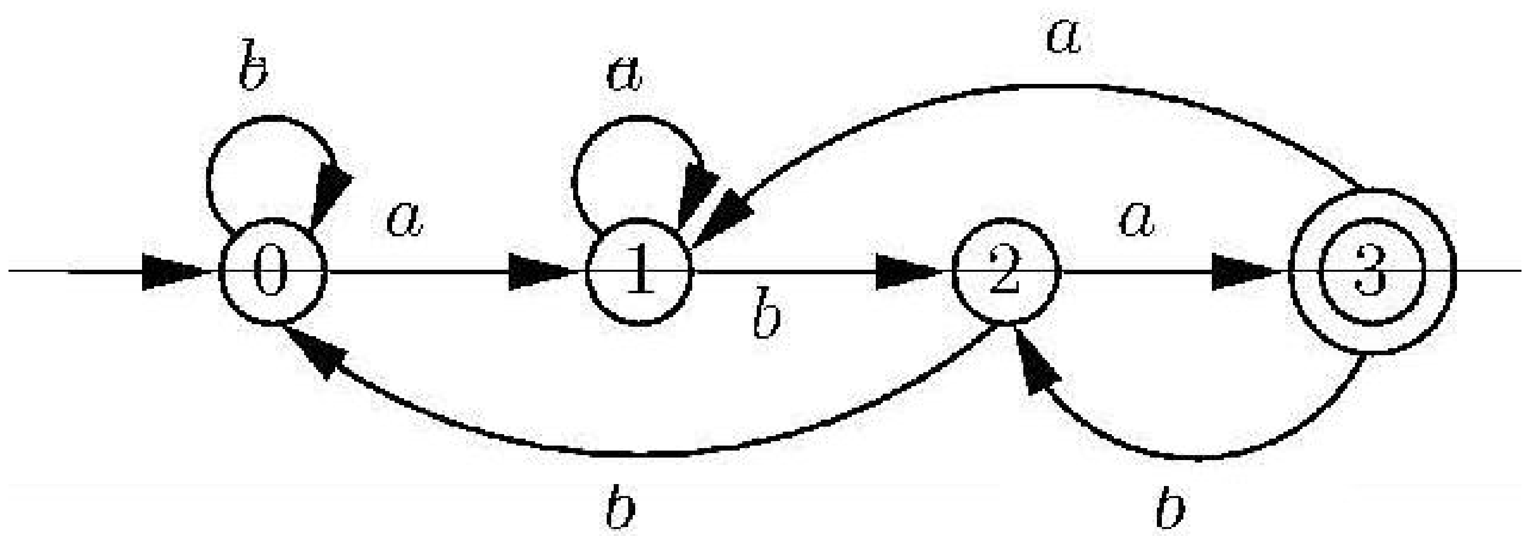
4) $q_0 \in Q$: estado inicial

3) δ : función de transición

$$\delta : Q \times \Sigma \rightarrow Q$$

5) $F \subseteq Q$: conjunto de estados finales (o de aceptación)

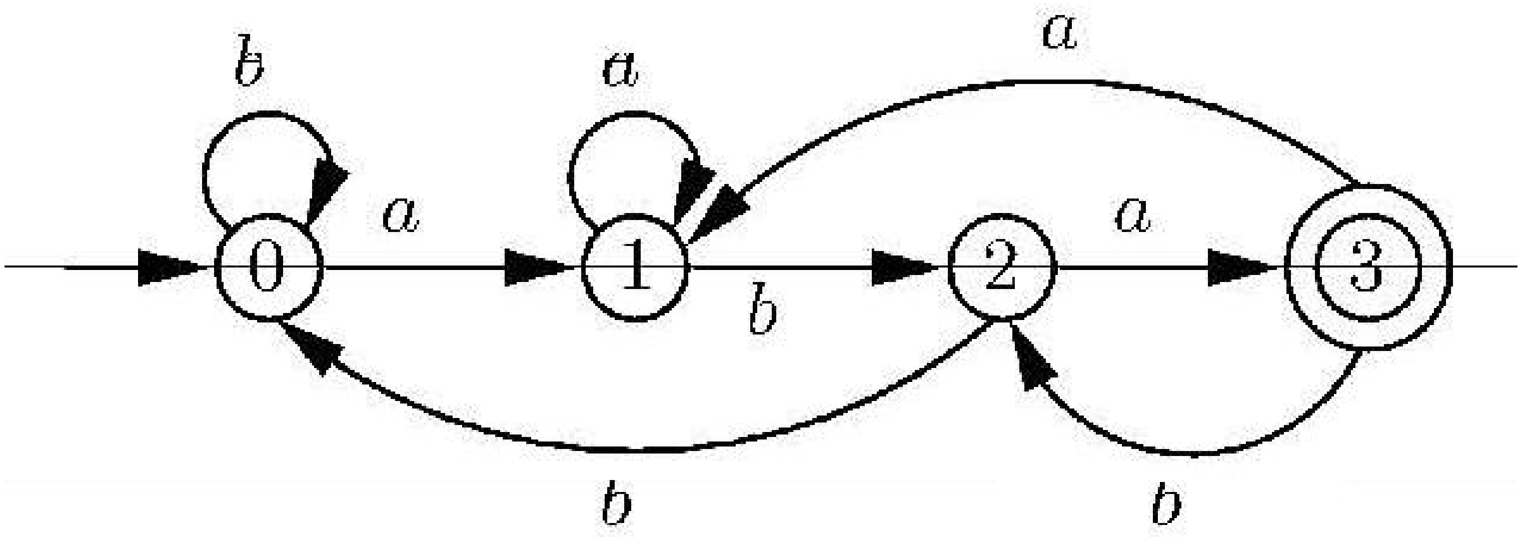
Ejemplo de autómata



String matching automata

- Dado un patrón $p=p_1. . .p_m$ construimos un autómata que acepte los textos con sufijo p (es decir, los textos que acaban en p)

Ejemplo p=aba



t= bababaa

Cada vez que encuentro aba llego al estado final 3

¿Cómo construir el autómata?

- Método inocente ($p = p_1 \dots p_m$)
  - Un estado por cada prefijo de p
  - Estado q_w quiere decir “w es el final del texto leído que es prefijo de p”
  - Para asignar $\delta(q_w,a)$ (w prefijo de p):
  - *i=1;* Si $y=w_i \dots w_{|w|}a$ *es prefijo de p*
    - $\delta(q_w,a)= q_y$
    - Si no i++ y repetir

String matching con autómata

StringMatching($t = t_1 \ldots t_n$, $p = p_1 \ldots p_m$)

Calcular M_p el autómata para p ($M_p=(Q, \Sigma, q_0, \delta, F)$)

$q= q_0$; I=vacío;

For i:=0 to n do

$q:=\delta(q,t_i)$

If esFinal(q) then I:=añadir(q,I)

Resultado I

Complejidad

- Construir el autómata (no visto): $O(|\Sigma| \cdot m)$
- String matching: $O(n)$

Enunciado del problema …

- Entrada: Dos strings $t= t_1 \ldots t_n$, $p= p_1 \ldots p_m$ sobre Σ
- Salida: El conjunto de posiciones de t donde aparece p, es decir, $I \subseteq \{1,\ldots, n-m+1\}$ tales que $i \in I$ sii $t_i \ldots t_{i+m-1} = p$

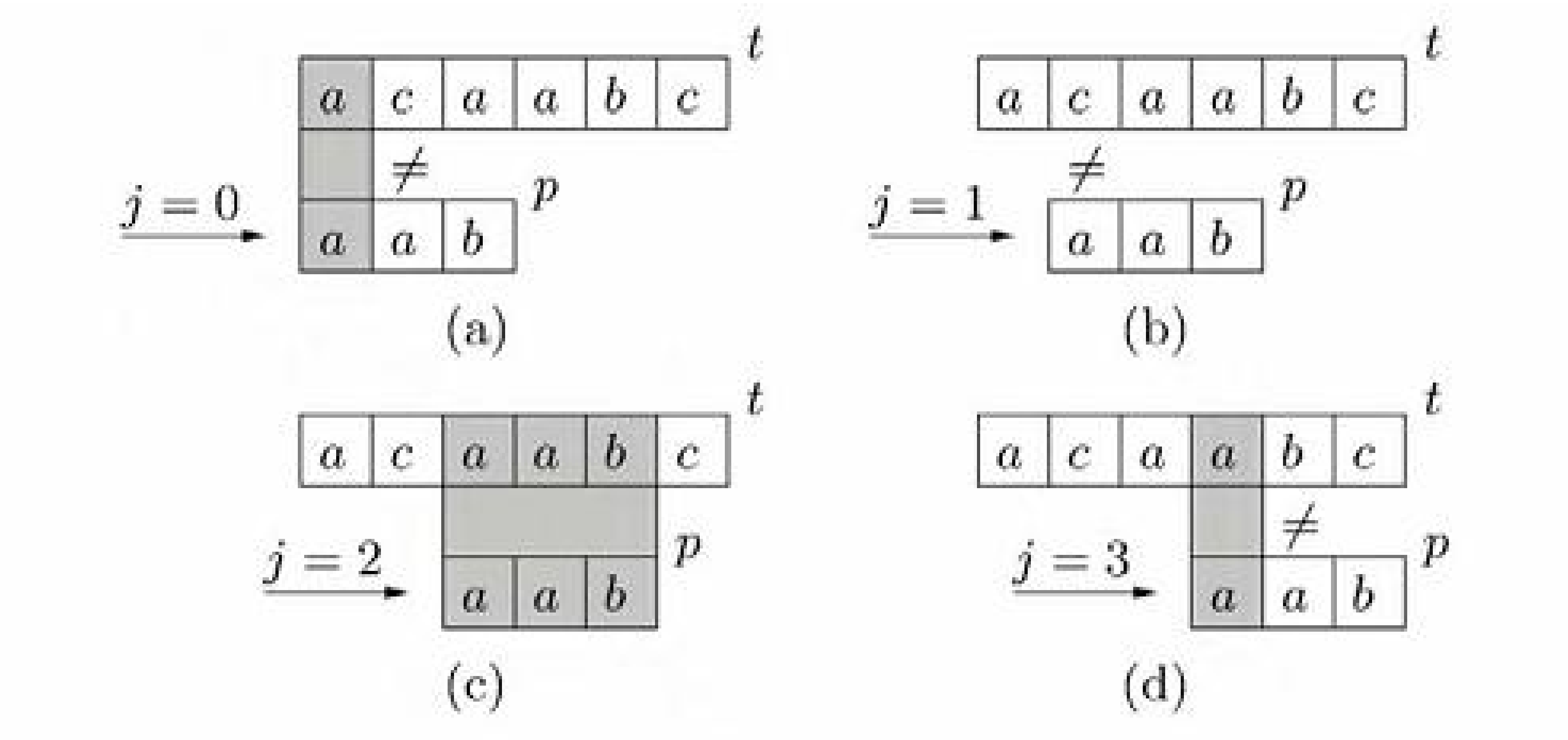
Vistos

- Algoritmo inocente: $O(m \cdot (n-m))$
- String matching autómata:
  - $O(|\Sigma| \cdot m)$ preprocesamiento del patrón
  - $O(n)$ string matching

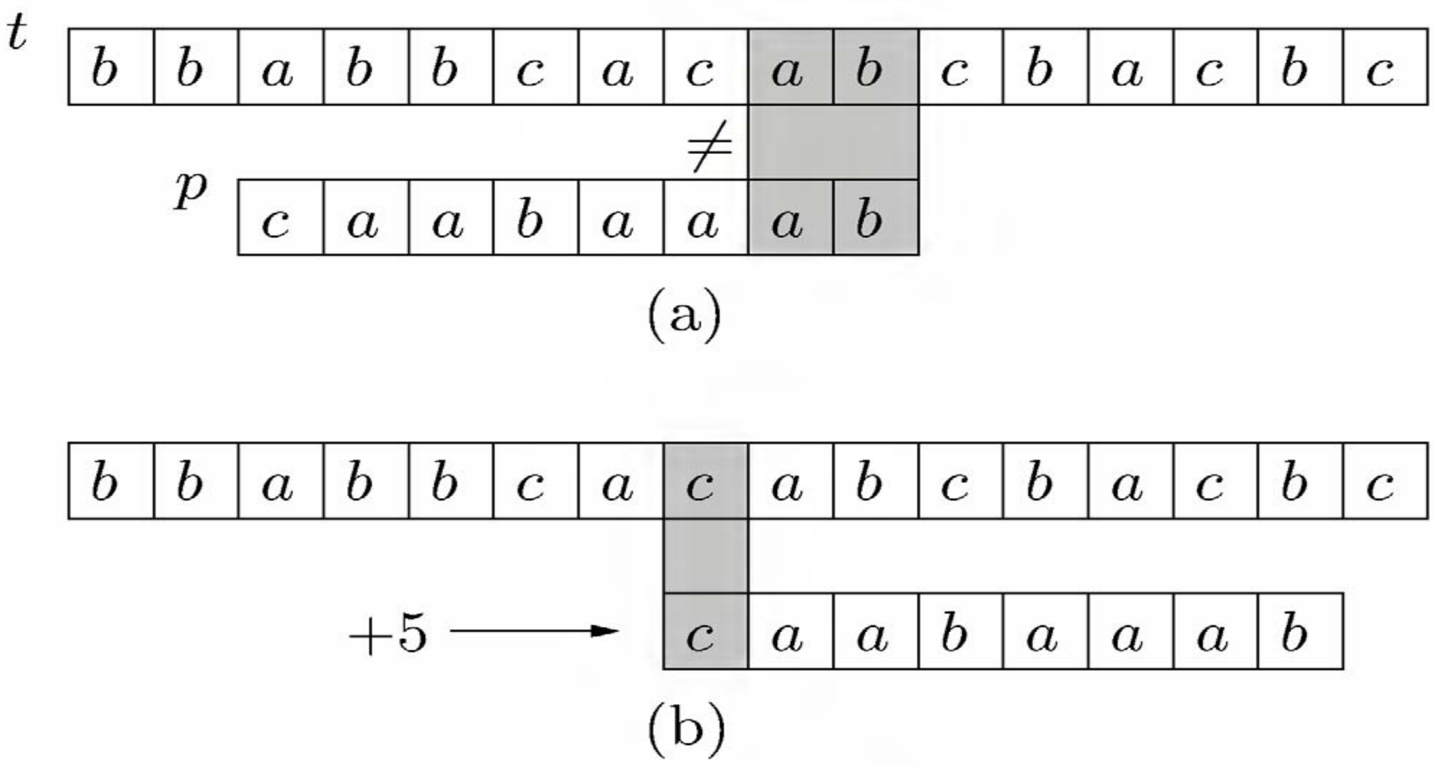
Algoritmo de Boyer-Moore

- Se trata de utilizar el algoritmo “inocente”: ir moviendo patrón sobre el texto
- Pero …
  - Cada comparación de $p_1 \ldots p_m$ con $t_{j+1} \ldots t_{j+m}$ la hacemos empezando por el final
  - Si podemos movemos más de una posición la j
- Usa preprocesamiento de p

Ejemplo del inocente …



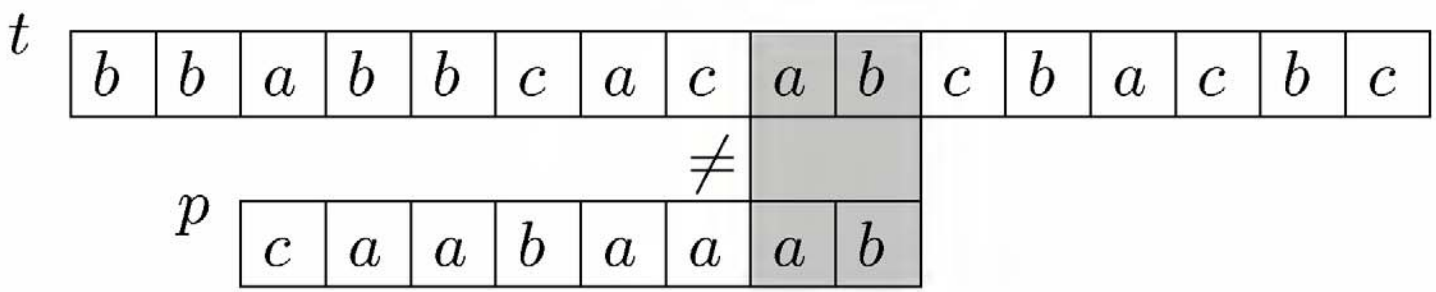
Boyer-Moore se comporta …



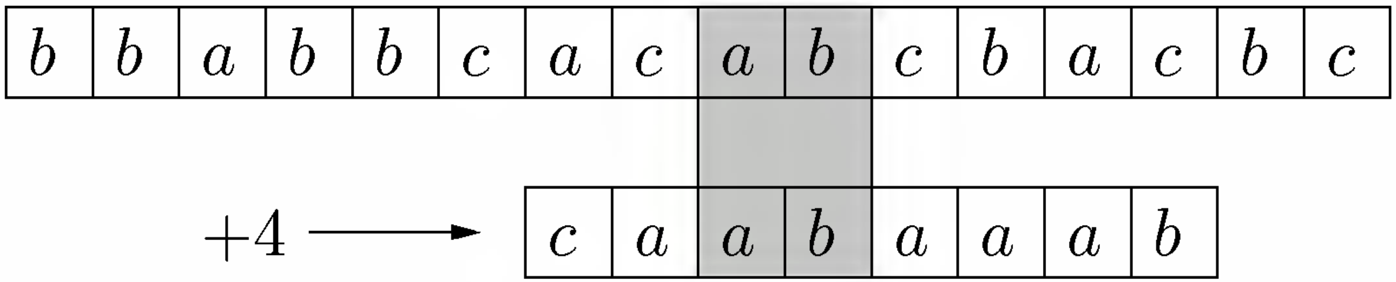
Regla del carácter raro

- Si c es la primera diferencia (carácter raro) entre t y p y está en la posición i de p
  - Busca la ocurrencia más cercana y anterior a i de c en p
- Tenemos ya precalculado cr(i,c) para cada i de 1 a m y para carácter c

Boyer Moore se comporta …



(a)



(c)

Regla del buen sufijo

- Si c es la primera diferencia (carácter raro) entre t y p y está en la posición i de p
  - Busca la ocurrencia de p[i+1..m] más cercana y anterior en p
  - Si no existiera prueba con p[j ..m] para j>i+1
- Tenemos ya precalculado bs(i) para cada i de 1 a m-1

Complejidad de Boyer-Moore

- Preprocesamiento $O(|\Sigma| \cdot m)$
- Caso peor: igual que el inocente $O(|\Sigma| \cdot m + n \cdot m)$
- Bastante rápido en la práctica
- Se puede mejorar el preproceso para que el caso peor sea $O(n+m)$

Comparando complejidades

Caso peor …

- Algoritmo inocente $O(m \cdot (n-m))$
- String matching automata $O(n+|\Sigma| \cdot m)$
- Boyer-Moore $O(n+m+|\Sigma|)$

El caso peor del Boyer-Moore ocurre poco

Alternativa: el Knuth-Morris-Pratt (mejor en el caso peor, peor en la práctica)

Separando el preprocesamiento …

- Algoritmo inocente: $O(m \cdot (n-m))$
- String matching autómata:
  - $O(|\Sigma| \cdot m)$ preprocesamiento del patrón
  - $O(n)$ string matching
- Boyer-Moore
  - $O(|\Sigma|+m)$ preprocesamiento del patrón
  - $O(n)$ string matching

Contenido

- El problema de string matching
- **Árboles de sufijos**
- Vectores de sufijos

Árboles de sufijos

- Se trata de preprocesar el texto
- Esto es útil cuando se quieren encontrar muchos patrones en el mismo texto (por ejemplo, muchos genes en el mismo DNA)

Árboles de sufijos

- El patrón p ocurre en t cuando p es el prefijo de un sufijo de t
- Se trata de calcular y almacenar eficientemente los sufijos de t

Ejemplo …

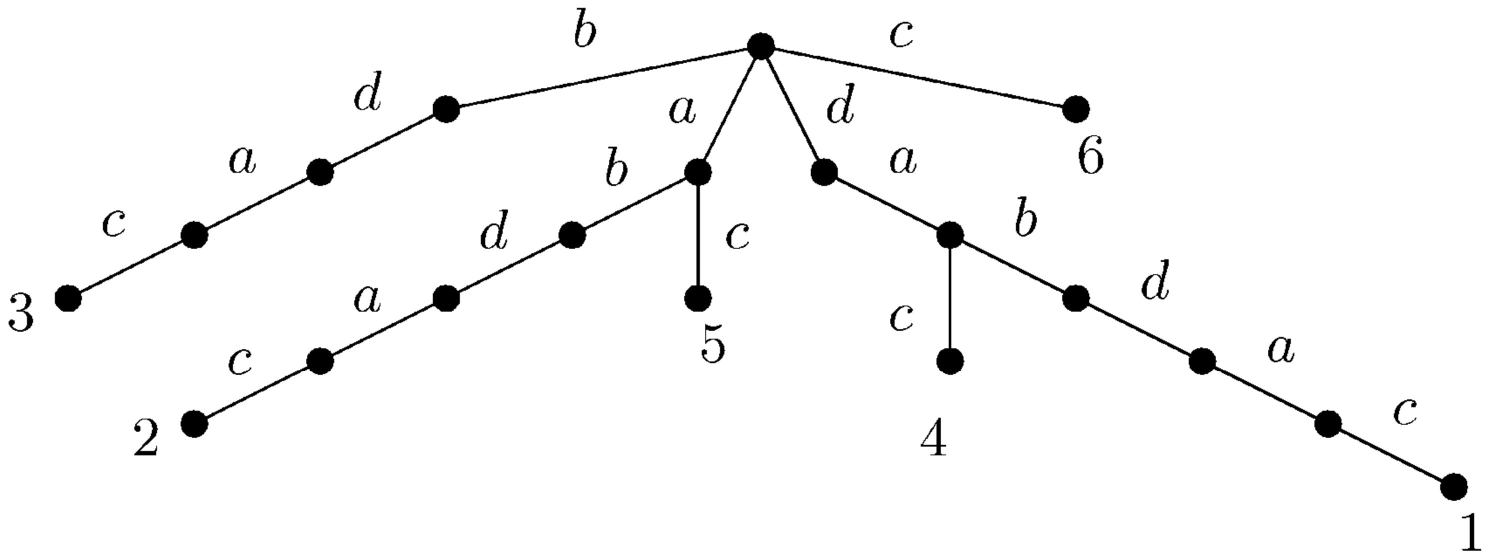


Fig. 4.5. A simple suffix tree for the text $t = dabdac$

Definición

- Dado un string $t=t_1 \ldots t_n$, un árbol sufijo simple es un árbol dirigido $T_t=(V,E)$ con una raiz r que cumple
  1. El árbol tiene n hojas con etiquetas 1, …, n. (La etiqueta i corresponde al sufijo $t_i \ldots t_n$)
  2. Las aristas del árbol están etiquetadas con letras del alfabeto
  3. Todas las aristas de salida de un nodo tienen letras diferentes
  4. El camino de la raiz a la hoja i tiene etiquetas $t_i \ldots t_n$

¿Funciona siempre?

- ¿Qué pasa con cadada?
- Hay sufijos que son prefijos de otros sufijos …
- Para evitarlo se construye el árbol de t$

Construcción simple

ÁrbolSufijos($t = t_1 . . . t_n$)

t':= t\$
T:= árbol con raiz r
for i:=0 to n do
{insertar $t_i . . . t_n$\$ en el árbol T}
Empezando en la raiz buscar un camino máximo $t_i . . . t_j$ terminando en el nodo x
Añadir al árbol los nodos correspondientes a $t_{j+1} . . . t_n$\$ como camino a partir de x con etiquetas $t_{j+1}, . . ., t_n$, \$, etiquetando la hoja con i

Resultado T (El árbol de sufijos de $t_1 . . . t_n$\$)

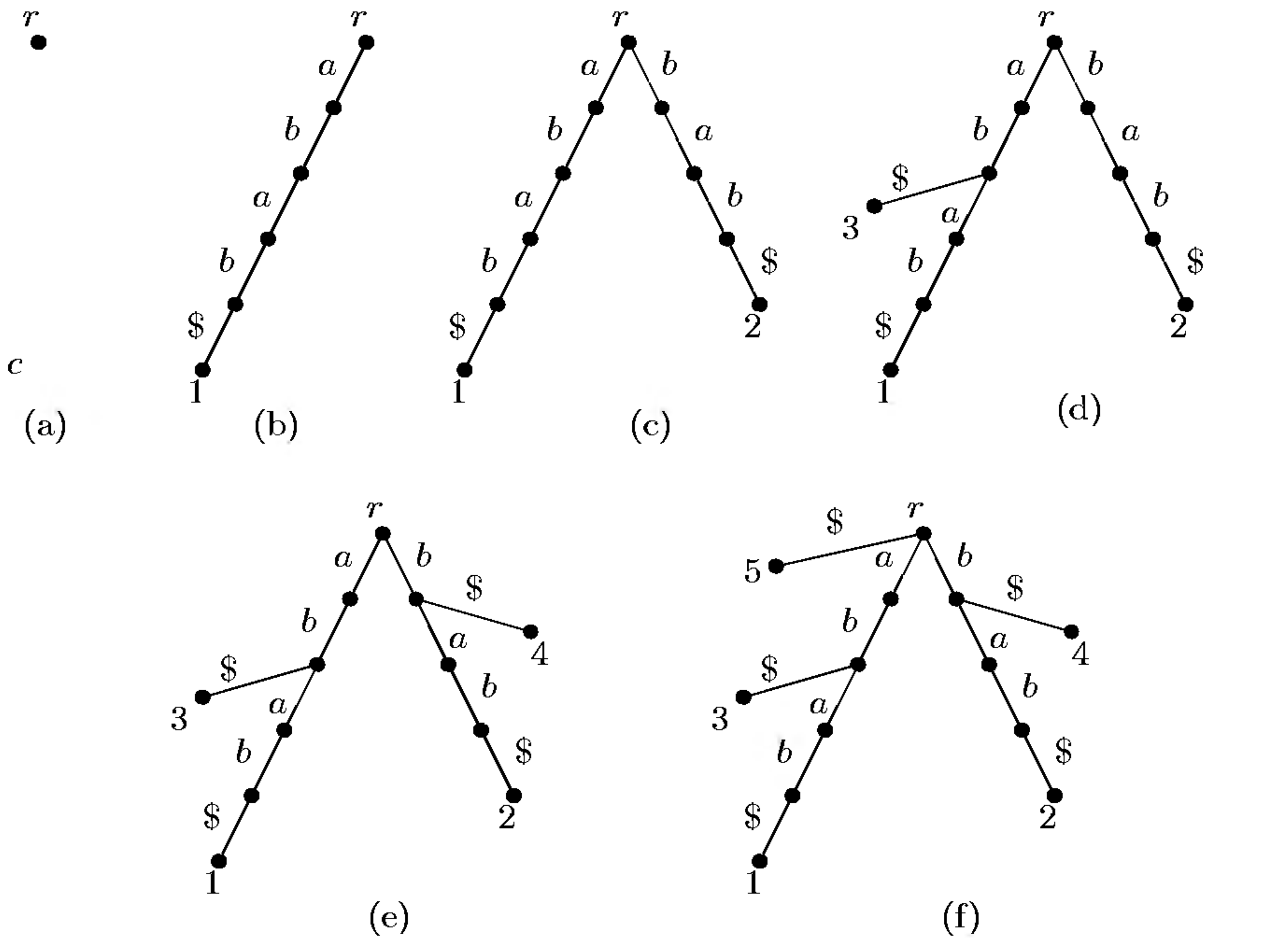


Fig. 4.6. The construction of a simple suffix tree for the text *abab*\$ using Algorithm 4.7

¿Cuánto cuesta construir el árbol?

- Con el algoritmo simple $O(n^2)$
- Se trata de añadir cada sufijo letra por letra

- Además ocupa bastante memoria, ¿es imprescindible?

String matching

- Si tenemos el árbol de sufijos de t, ¿cuánto cuesta encontrar si p es substring de t?
- ¿y todas las apariciones de p como substring de t?

String matching

- Saber si p está en t cuesta tiempo O(|p|)
- **Suponiendo que ya tenemos el árbol**
- Encontrar todas las ocurrencias de p en t puede ser muy largo dependiendo del tamaño del subárbol con raiz p

- Ejemplo: $t=a^nb^nc$

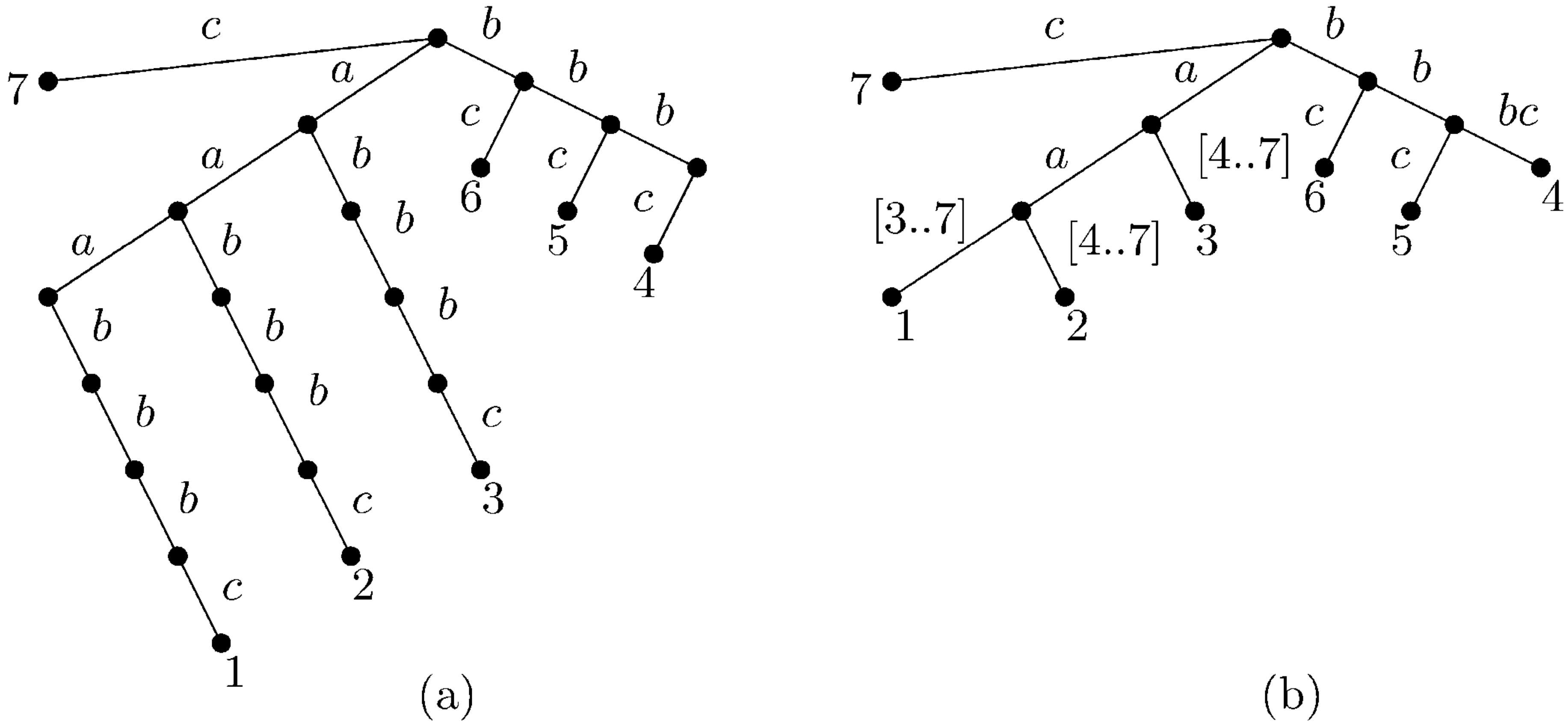
¿Podemos mejorar?

- Vamos a buscar una representación compacta del árbol
- Ahorraremos memoria y tiempo de string matching

Árboles de sufijos compactos

- Podemos eliminar los nodos con un solo hijo … poniendo strings como etiquetas
- No tenemos que escribir las etiquetas largas

- En lugar de una etiqueta de k símbolos (ocupa $k \cdot \log(|\Sigma|)$) usamos los dos índices en el texto (ocupa $2 \cdot \log n$)



c
b
7
a
b
a
b
c
b
c
6
5
c
4
a
b
b
b
b
b
c
b
b
c
3
2
c
1
(a)
c
b
7
a
b
bc
c
[4..7]
a
6
c
4
[3..7]
[4..7]
3
5
1
2
(b)

Definición formal

- Dado un string $t=t_1\ldots t_n$, un árbol sufijo compacto es un árbol dirigido $T_t=(V,E)$ con una raiz r que cumple
  1. El árbol tiene n hojas con etiquetas 1, …, n. (La etiqueta i corresponde al sufijo $t_i\ldots t_n$)
  2. Cada vértice interior tiene al menos **dos hijos**
  3. Las aristas del árbol están etiquetadas con substrings de t (representadas tal cual o con los índices de inicio y final en el texto)
  4. Todas las aristas de salida de un nodo empiezan por letras diferentes
  5. El camino de la raiz a la hoja i tiene etiquetas $t_i\ldots t_n$

Tamaño de un a. sufijo compacto

- Como tiene n hojas y todos los nodos interiores tienen al menos 2 hijos ...
  - Hay un máximo de n-1 nodos interiores
  - En total 2n-1 nodos
- Codificar un nodo cuesta como máximo 2 logn bits
- En total O(n logn)

Notación en el algoritmo

- Si x es un nodo del árbol sufijo compacto:
  - pathlabel(x) es la concatenación de las etiquetas desde la raiz a x
  - depth(x) = |pathlabel(x)|
- Si x es un nodo del árbol sufijo:
  - Pos(x) es la mínima etiqueta de una hoja del subárbol de raiz x

Construcción compacta (1)

ÁrbolSufijosCompacto($t = t_1. . .t_n$)

```
T:= ÁrbolSufijos(t);

{Eliminar los nodos de grado 2}
X:= nodos de grado 2 de T;
While not EsVacío(X) do
        Elegir x en X con hijo z, padre y
        Reemplazar (y,x), (x,z) por (y,z) con etiqueta
                label(y,z) = label(y,x) label(x,z)
        Borrar x
```

…

Construcción compacta (2)

ÁrbolSufijosCompacto($t= t_1. . .t_n$)

…

{Comprimir las etiquetas largas}
For (x,y) arista do
If label(x,y) >= log(n-|Σ|) then
{Reemplazar la etiqueta por los números de posición}
label'(x,y) = (Pos(y)+depth(x), Pos(y)+depth(x)+|label(x,y)|-1);

label=label'

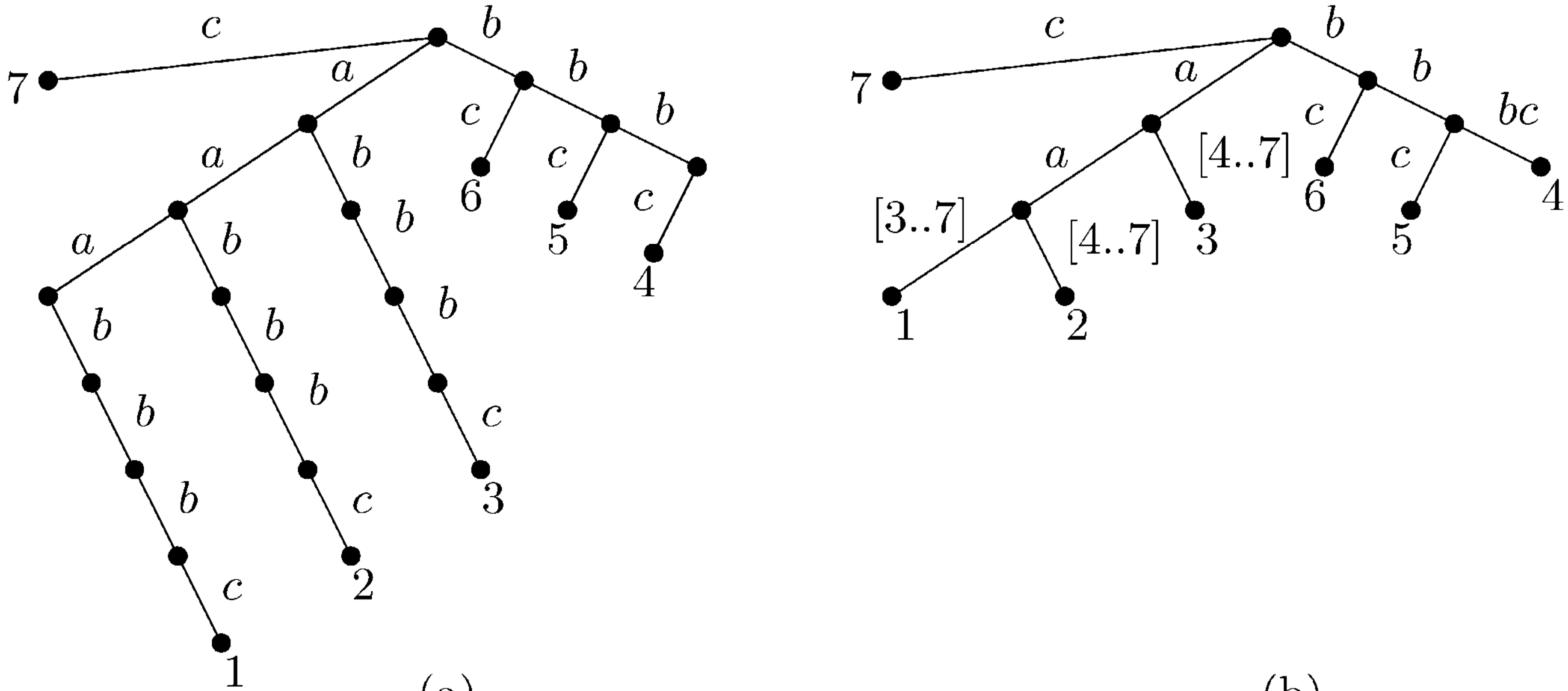
Resultado T (El árbol de sufijos compacto de $t_1. . .t_n\$$)

Coste de la construcción

- Como el árbol sufijo inicial puede ser de tamaño $O(n^2)$ el tiempo es $O(n^2)$
- Hay métodos de construcción del árbol sufijo compacto sin pasar por el inicial que tardan O(n log n) (algoritmo de Ukkonen)
- Como el tamaño del árbol es O(n log n) esto es lo mínimo posible ...

String matching usando árboles de sufijos compactos

- Buscamos un patrón p en un texto t
- Tenemos que encontrar un nodo x del árbol de forma que el camino de la raiz a x sea p ó bien tenga p como prefijo
- Las apariciones de p en t nos las dan las hojas del subárbol de x



(a) (b)

StringMatching(1)

StringMatching(t= t_1 … t_n, p= p_1 … p_m)

(1) T:= ÁrbolSufijosCompacto (t);

(2) {Inicialización}
x:= raiz(T); {vértice actual}
i:=1; {posición actual en *p*}
found:=false;
possible:=true;

…

StringMatching(2)

While not found and possible

(3a) {buscar una arista desde x con label empezando en p_i}

fitting:=false;

U:= hijos de x;

While not fitting and not esVacío(U) do

Elegir v en U

If label(x,v)= p_i a then

fitting:=true

mylabel:=label(x,v)

else if label(x,v)=[k..l] and $t_k = p_i$ then

fitting:=true

l'=min(l, k+m-i)

mylabel:= $t_k \dots t_{l'}$

{leer la parte necesaria de la etiqueta}

borrar(v,U)

StringMatching(3)

While not found and possible

….

(3b) {Comparar mylabel con la parte de p por encontrar}

If (p_i … p_m no es prefijo de mylabel) and(mylabel no es prefijo de p_i … p_m) then

possible := false {p no aparece en t}

else if mylabel es prefijo de p_i … p_m then

x:=v

i:=i+|mylabel|

else {p_i … p_m es prefijo de mylabel}

x:=v

found := true

(4) if found then

Calcular el conjunto I de las etiquetas de hojas en el subárbol de raiz x (búsqueda en profundidad)

Resultado *I* (El conjunto *I* de posiciones, donde empieza una ocurrencia de p como substring de t)

Complejidad

3(a)

- Si sólo hay que atravesar etiquetas no comprimidas (abab) para encontrar un patrón $p=p_1\ldots p_m$ basta tiempo $|\Sigma|m$
- Si hay que pasar por alguna etiqueta "comprimida" [2..5] eso quiere decir que la etiqueta corresponde a un fragmento largo (si no no se hubiera comprimido), a un fragmento de tamaño $\Omega(\log n)$. Luego eso sólo puede ocurrir $m/\log n$ veces
- Leer cada fragmento comprimido cuesta como mucho $|\Sigma|\log n$

Complejidad

- 3(b) cuesta tiempo m en total
- Atravesar el subárbol encontrado cuesta O(k) si p ocurre k veces en t

- El algoritmo completo cuesta

  $O(n \log n + m |\Sigma| + k)$

Resumen string matching

Encontrar todas las ocurrencias de …
un patrón de tamaño m en un texto de tamaño n

- Algoritmo inocente $O(m\cdot(n-m))$
- String matching automata $O(n+|\Sigma|\cdot m)$
- Boyer-Moore $O(n+m+|\Sigma|)$
- Árboles de sufijos compactos (k es el número de ocurrencias) $O(n \log n + m\,|\Sigma|+k)$

Resumen …

- Algoritmo inocente: $O(m \cdot (n-m))$
- String matching autómata:
  - $O(|\Sigma| \cdot m)$ preprocesamiento del patrón
  - $O(n)$ string matching
- Boyer-Moore
  - $O(|\Sigma|+m)$ preprocesamiento del patrón
  - $O(n)$ string matching
- Árboles de sufijos
  - $O(n \log n)$ preprocesamiento del **texto**
  - $O(m\ |\Sigma|+k)$ string matching

Resumen

Encontrar todas las ocurrencias de …
un patrón de tamaño m en **N textos** de tamaño n

- Algoritmo inocente $O(N \cdot m \cdot (n-m))$
- String matching automata $O(N \cdot n+|\Sigma| \cdot m)$
- Boyer-Moore $O(|\Sigma| \cdot m+n \cdot m \cdot N)$, $O(n \cdot N+m+|\Sigma|)$
- Árboles de sufijos compactos (k es el número máximo de ocurrencias)
  $O(N \cdot n \cdot \log n + N \cdot m \cdot |\Sigma| + N \cdot k)$

Lo mejor: preprocesamiento de patrón (autómata o BM)

Resumen

Encontrar todas las ocurrencias de …

M patrones de tamaño m en un texto de tamaño n

- Algoritmo inocente $O(M \cdot m \cdot (n-m))$
- String matching automata $O(n \cdot M+|\Sigma| \cdot m \cdot M)$
- Boyer-Moore $O(|\Sigma| \cdot m \cdot M+n \cdot m \cdot M)$, $O(n \cdot M+m \cdot M+|\Sigma|)$
- Árboles de sufijos compactos (k es el número total de ocurrencias) $O(n \log n + m \cdot |\Sigma| \cdot M +k)$

Lo mejor: preprocesamiento de texto (árboles de sufijos)

Contenido

- El problema de string matching
- Árboles de sufijos
- **Vectores de sufijos**

Vectores de sufijos

- Hemos visto la utilidad de los árboles de sufijos
- Ahora nos planteamos guardar los sufijos en un vector, ordenados alfabéticamente (lexicográficamente)

- Ejemplo: s=ababbabbb

Definición

- El vector de sufijos de un string s es

$$A(s)=(j_1, \ldots, j_n)$$

tal que el orden alfabético de los sufijos es $s[j_1,n] < s[j_2,n] < \ldots < s[j_n,n]$

String matching con vectores de sufijos

- Para encontrar todas las ocurrencias de un patrón p en un texto t contando con el vector de sufijos de t
- Usar búsqueda dicotómica para encontrar el primer y último sufijo de t que empiezan por p

Algorithm 4.12 String matching using a suffix array

Input: A text t and a pattern p over an ordered alphabet Σ, and a suffix array $\mathcal{A}(t)$ for the text t.

1. Use binary search to find the first position i and the last position j in the suffix array such that p starts as a substring in t at positions $\mathcal{A}(t)[i]$ and $\mathcal{A}(t)[j]$.
2. $I := \{\mathcal{A}(t)[i], \mathcal{A}(t)[i+1], \ldots, \mathcal{A}(t)[j]\}$.

Output: The set I of all positions, where p starts as a substring in t.

Coste: O(m log n +k) donde k es el número de ocurrencias

¿Cuánto cuesta construir el vector?

- Se usa el algoritmo “skew” o sesgo basado en ordenación
- Cuesta tiempo O(n)

- Veamos las ideas prales …

Radix sort para strings

- Podemos ordenar alfabéticamente n strings de longitud d sobre un alfabeto de k símbolos en tiempo O((n+k)d)
- Para ello ordenamos de forma estable según cada una de las posiciones de la 1 a la d

Ordenar n valores de 0 a k

Algorithm 4.13 Counting Sort

Input: An array $A = (A(1), \ldots, A(n))$ of integers from the range $\{0, \ldots, k\}$.

1. Counter initialization:

   for $i := 0$ **to** k **do**
   $\quad c(i) := 0$

2. Count the number of elements of each type:

   for $j := 1$ **to** n **do**
   $\quad c(A(j)) := c(A(j)) + 1$

3. Count the number of elements less or equal to i:

   for $i := 1$ **to** k **do**
   $\quad c(i) := c(i) + c(i-1)$

4. Calculate the position of each element in the sorted array:

   for $j := n$ **downto** 1 **do**
   $\quad B(c(A(j))) := A(j)$
   $\quad c(A(j)) := c(A(j)) - 1$

Output: The sorted array $B = (B(1), \ldots, B(n))$.

O(n+k)

Para ordenar n strings …

- El algoritmo anterior ordena **de forma estable** n datos de 0 a k
- Lo utilizamos para ordenar, según una posición, n strings sobre un alfabeto de k símbolos

Algorithm 4.14 Radix Sort

Input: An array $A = (a_1, \ldots, a_n)$ of strings of length d in k-letter alphabet

for $i := 1$ **to** d **do**

Sort A according to the i-th letter using Algorithm 4.13.

Output: The sorted array.

$O((n+k)d)$

El algoritmo para construir el vector

- Separa los sufijos que tienen longitud múltiplo de 3 (S^0) del resto ($S^{1,2}$)
- Construye primero $A^{1,2}$, el vector de $S^{1,2}$ ordenando sólo las 3 primeras letras de cada sufijo por radix-sort, y si no es suficiente se cambia el alfabeto (cada tres letras viejas es una nueva) y se hace llamada recursiva
- Construye A^0 usando $A^{1,2}$
- Mezcla A^0 y $A^{1,2}$

- Coste O(n log n)

Algorithm 4.15 Skew algorithm for constructing a suffix array

Input: A string $s = s_1 \ldots s_n$ over the ordered alphabet $\{1, \ldots, n\}$.

1. Initialization:
   a) Define $s_{n+1} := 0$, $s_{n+2} := 0$, and $s_{n+3} := 0$, and let s denote the string $s_1 \ldots s_{n+3}$.
   b) Let $S_i := s_i \ldots s_n$ be the i-th suffix of s for all $0 \leqslant i \leqslant n+1$.
   c) Let $k_i = \max\{k \leqslant n+1 \mid k \equiv i \pmod 3\}$, for $i \in \{1,2\}$;
   let $l_i := |\{k \leqslant n+1 \mid k \equiv i \pmod 3\}|$, for $i \in \{1,2\}$.
2. Construct the suffix array $\mathcal{A}^{1,2}$ for the set $\mathcal{S}^{1,2}$ of all suffixes S_i such that $i \not\equiv 0 \pmod 3$:
   a) Let $t_i := s[i, i+2]$ for all $0 \leqslant i \leqslant n+1$ such that $i \not\equiv 0 \pmod 3$.
   b) Sort the triples t_i, for all $i \leqslant \max\{k_1, k_2\}, i \not\equiv 0 \pmod 3$, using radix sort (Algorithm 4.14).
   c) Assign lexicographical names $t'_i \in \{1, \ldots, \lceil \frac{2}{3}(n+1) \rceil\}$ to the triples, i.e., define the t'_i such that $t'_i = t'_j$ if $t_i = t_j$, and $t'_i < t'_j$ if $t_i \prec_{\text{lex}} t_j$.
   d) If all t'_i are distinct, construct $\mathcal{A}^{1,2}$ directly from the order of the t_i; else, recursively compute the suffix array $\tilde{\mathcal{A}}$ for the string

$$\tilde{s} = \tilde{s}_1 \ldots \tilde{s}_{l_1+l_2} := t'_1 t'_4 t'_7 \ldots t'_{k_1} \cdot t'_2 t'_5 t'_8 \ldots t'_{k_2}$$

   and construct $\mathcal{A}^{1,2}$ from $\tilde{\mathcal{A}}$ by substituting the indices of the t'_i for the indices of the $\tilde{s}_j$.

3. Construct the suffix array $\mathcal{A}^0$ for the set $\mathcal{S}^0$ of all suffixes S_i such that $i \equiv 0 \pmod 3$:
   a) Represent S_i by the pair (s_i, S_{i+1}) for all $1 \leqslant i \leqslant n$ such that $i \equiv 0 \pmod 3$.
   b) Consider the order of $\mathcal{S}^0$ as given by the order of the second component of its elements in $\mathcal{A}^{1,2}$, and sort $\mathcal{S}^0$ by counting sort (Algorithm 4.13) with respect to the first components.
4. Merge the two suffix arrays $\mathcal{A}^{1,2}$ and $\mathcal{A}^0$ into a suffix array $\mathcal{A}(s)$.

Output: The constructed suffix array $\mathcal{A}(s)$.
